TRANSFORMATION AND BENCHMARK EVALUATION FOR SQL QUERIES

by

Eric Qian Wu

B.Sc. Peking University, Beijing, China, 1986

M.Sc. Peking University, Beijing, China, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in the School
of
Computing Science

ISBN 0-315-78200-5

APPROVAL

Name:	Eric Qian Wu
Degree:	Master of Science
Title of thesis:	**Transformation and Benchmark Evaluation for SQL Queries**

Examining Committee: Dr. Veronica Dahl, Chairman

Dr. Nick Cercone, Senior Supervisor

Dr. Jiawei Han, Senior Supervisor

Dr. Fred Popowich, External Examiner

Date Approved: November 29, 1991

Title of Thesis/Project/Extended Essay

Transformation and Benchmark Evaluation for SQL Queries.

Author: ______________________________
(signature)

Eric Qian Wu
____
(name)

December 10, 1991
____
(date)

ABSTRACT

Database query optimization research has been ongoing for a long time. Nevertheless considerable performance deviations persist between retrieval times for different, but logically equivalent, expressions of SQL queries. It would appear that in many actual applications the query optimizer cannot efficiently optimize the query with respect to retrieval time unless query transformation and the physical (index) structure of the database are taken into account. In this thesis an experimental performance study is carried out, with the help of the Wisconsin Benchmark, to test which kinds of queries are generally more efficient than other logically equivalent queries (based on our classification of SQL queries). This research is intended to provide an aid for use in natural language database interfaces where automatic SQL query generation results in more efficient query transformations to optimize subsequent data retrieval.

To my mom and dad

ACKNOWLEDGMENTS

I would like to thank my two senior supervisors, Dr. Nick Cercone and Dr. Jiawei Han for their generosity, guidance, especially their extraordinary patience while modifying and editing the thesis drafts, which make all things possible for me. I am also grateful to Dr. Fred Popowich for his careful reading and valuable comments. My sincere thanks also go to Mr. Gary Hall who talked with me, helped me to finish the testing, read through my draft very carefully and gave me very important suggestions. I would also like to acknowledge the financial support from the School of Computing Science and the Center for Systems Science, Simon Fraser University.

My thanks also go to my parents for their love, education and encouragement in so many years, to my two brothers and best friends who made my first two years in Canada memorable.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

One of the most appealing properties of many relational database systems is their nonprocedural user interface. Users specify only what data is desired, leaving the system optimizer to choose how to access that data. The built-in decision capabilities of the optimizer therefore play a central role regarding system performance. Automated selection of optimal access plans is a rather difficult task, because, even for simple queries, there are many alternatives and factors affecting each query's performance. Critics of relational systems point out that their nonproceduralty prevents users from navigating through the data in the ways they believe to be the most efficient. Developers of relational systems claim that systems could be capable of making very good decisions about how to perform users' requests based on statistical models of databases and formulas for estimating the costs of different execution plans. Optimizer effectiveness in choosing efficient execution plans is critical to system response time. For example, in an experiment described later in the thesis, a query was executed in 41.42 seconds on our Oracle system[1] while the response time of another query which produces exactly the same answer was 74.3 minutes, about 108 times longer. In the case where the first query is to be used interactively, the second one cannot be justified since it is very inefficient. The same performance problem arises for a number of equivalent SQL queries. This huge performance difference is due to the fact that it is usually very difficult for the optimizer to know the nature of some complicated queries. Thus, the database query optimizer chooses totally different algorithms for the evaluation of queries which request the same data in different ways. Optimization algorithms are built-in for only certain kinds of queries, thus the optimizer sometimes cannot find the most efficient evaluation algorithm. Therefore, it is important to transform, if possible, those complicated queries into simpler ones in order to make use of built-in optimization algorithms.

Generally speaking, the problems we encounter include: what kinds of SQL queries are more efficient than others in particular situations? For those queries which will produce the same results, is there any algorithm to transform one query into another equivalent query in order to take the advantage of built-in query processing algorithms? For query evaluation, what roles do the indexes play? Does the use of an index always result in better time efficiency?

This thesis is organized into six chapters. After motivating and introducing the

[1]Oracle RDBMS, Version 6.1.

problems in Chapter 1, a classification of SQL queries is given in Chapter 2 in order to establish the foundation for the transformation and performance experiment later on. Chapter 3 illustrates some specific transformation algorithms and their theoretical cost models. An actual testing environment is established in Chapter 4 from which the testing results are driven. A more general transformation algorithm is given in Chapter 5. Chapter 6 concludes that the transformation algorithms are practical and effective, thus a better retrieval performance can be obtained from those transformations.

CHAPTER 2

A Classification of SQL Queries

SQL is a block-structured, database interface language which has been implemented in many commercial relational database systems, e.g., the SEQUEL system, System R, DB2 and Oracle. The principal advantage of the relational data model is that it allows a user to express the desired results of a query in a high-level nonprocedural data language without specifying the access paths to stored data. Relational calculus and relational algebra were designed to concisely specify a complex query to a database. However, the mathematics of the relational system is difficult for normal users to grasp, thus their use as data languages to access an actual database may be limited. As a result, SQL was developed as an interface language. SQL is as powerful as the relational calculus and the relational algebra in the sense that SQL exhibits the major expressive power implicit in the relational calculus and algebra, but possesses the additional features like readability, which makes SQL easier for nontechnical users to learn and use. One of the more interesting features of SQL is the capability of nesting query blocks to an arbitrary depth. Without this capability, the power of SQL is severely restricted. However, techniques which have been used to implement this feature in existing systems are, in general, inefficient and, in view of the popularity of SQL-like data languages, it is imperative to develop efficient methods for processing nested queries. Our classification of SQL queries will be based on query blocks and the relationship between those query blocks.

A *query block* is represented by a SELECT clause, a FROM clause, and zero or one WHERE clause. The SELECT clause specifies the columns of the tables to be output and operations on the columns. Aggregate functions can be used as an operation on a column of a table. These aggregate functions in SQL are *SUM, AVG, COUNT, MAX* and *MIN.* The FROM clause specifies the tables referenced. The WHERE clause specifies the predicates which tuples of the tables indicated in the FROM clause must satisfy. Different predicates decide the different relationships between related query blocks.

A sample *predicate* is of the form [$R_i.C_k$ **op** X], where R_i is a table name, C_k is the column name of a table, $R_i.C_k$ represents the column C_k of the table R_i. X is a constant or a list of constants, and **op** is a scalar comparison operator (=, !=, >, >=, <, <=) or a set membership operator (IN, NOT IN). As long as the predicates in the WHERE clause are restricted to the simple predicates of the form [$R_i.C_k$ **op** X], only single-table queries can be formulated. For more general queries, the simple predicates may be extended in three ways.

1. Nested predicate: X may be replaced by Q, an SQL query block, to yield a

predicate of the form [$R_i.C_k$ **op** Q]. The **op** may be a scalar comparison or set membership operator. This predicate form implies that the subquery Q must result in a single-column table.

2. Join predicate: X may be replaced by $R_j.C_h$, to yield a predicate of the form [$R_i.C_k$ **op** $R_j.C_h$], where R_j is a table name, C_h is a column name in the table R_j. The **op** is a scalar comparison operator.

3. Division predicate: $R_i.C_k$ and X may be replaced by two query blocks Q_1 and Q_2, respectively, to yield a predicate of the form [Q_1 **op** Q_2]. The **op** may be a scalar comparison operator, set membership operator or set comparison operator (=, !=, CONTAINS, NOT CONTAINS).

Among predicates, the nested predicate and the join predicate are more interesting to us since no actual database system implements the division operation[2]. Thus, the division predicate is not considered further in our performance testing. The classification of SQL queries in this thesis is based on those predicates.

For demonstration purposes, we assume the following tables:
S(*sno*, sname, budget, city)—-the Supplier table
P(*pno*, pname, color, weight, city)—-the Part table
SP(*sno, pno*, qty, destination)—-the Shipment table

[2] *CONTAINS* and *NOT CONTAINS* are the two division operators in SQL language (See page 78 in [17]). But no actual RDBMS system supports these operators.

An S tuple contains the number (identifier), name, budget and location of a supplier. Each P tuple contains the number (identifier), name, color, weight and storage location of a part. Each SP tuple has fields for a supplier number, a part number, the quantity of parts the supplier supplies and the destination city for the shipment. Among these attributes of each table, italicized attribute(s) means the primary key(s) for that table.

2.1 Single-Block Queries

A *single-block* SQL query contains only one query block. Based on the number of tables involved, single-block queries can be of two kinds.

2.1.1 Single-Table Queries

A *single-table* query retrieves information from only one table. The number of predicates in the WHERE clause can be greater than one, of course. For example, find the number of the supplier whose name is 'Simon Fraser' in Burnaby:

```
SELECT   sno
FROM     S
WHERE    sname = 'Simon Fraser'  AND
         city = 'Burnaby'.
```

The evaluation of this kind of query is fairly straight forward. Most current database optimizers can take the advantage of available index information in order to find efficient ways for evaluation.

2.1.2 Multi-Table Queries (Join Queries)

A *multi-table* query retrieves information from two or more tables and must specify some join predicates. Thus, a *single-block multi-table query* is also called *join query*. For example, find supplier numbers and numbers of red parts which are stored at cities where there are some suppliers:

```
SELECT   sno, pno
FROM     S, P
WHERE    P.color = 'Red' AND
         S.city = P.city.
```

For a join query, the Oracle optimizer considers both the nested-iteration method and the merge-sort join method, as well as all of the possible 'reasonable' orders in which tables may be scanned, possibly with some available indexes. Whereas the nested-iteration method of joining two tables requires one table to be retrieved as many times as there are tuples that satisfy predicates on the other table, the merge-sort join method requires both tables to be simultaneously retrieved only once, provided that the tables are first sorted in join-column order. The nested-iteration method is better if one of the tables is small enough to fit into the main memory cache. In that case, this table may be the inner loop table, thus less disk I/O occurs. However, if all tables are too large to fit into the main memory cache, the merge-sort join method performs better.

2.2 Nested Queries

A *nested query* includes subqueries. A *subquery* is a query block that is used in a clause of a higher level SQL statement. A query block comprising a subquery is called an *inner block*; a query block containing a subquery is called an *outer block*. A table in the FROM clause of an inner block is called an *inner table*; a table in the FROM clause of an outer block is called an *outer table*. In the WHERE clause of a query block, there may be several subqueries connected by SQL logical operators *AND* or *OR*, and subqueries may also be nested with depth of greater than one. These would form more general, thus more complicated, nested queries. Unless otherwise specified, all the nested query examples in this chapter are the simplest nested queries, which means there is only one inner block and one outer block in each query and there is only one table in each query block.

2.2.1 TypeA Nesting

A nested query is a *TypeA* (*A* means *Aggregate*) query if the inner query block Q does not contain a join predicate that references any outer table, and if the SELECT clause of Q consists solely of an aggregate function over a column of an inner table. For example, find the numbers of suppliers who ship parts with the maximal part number:

```
SELECT   sno
FROM     SP
```

```
WHERE   pno =
    (   SELECT  MAX( pno )
        FROM    P ).
```

Query blocks can be join blocks, which means that the join predicates are permitted within one query block. Thus, the following query is also a TypeA query: find the name of a supplier with the largest supplier number who ships more than 100 pieces of any kind of parts to the city where a supplier stays.

```
SELECT  sname
FROM    S
WHERE   sno =
    (   SELECT  MAX( sno )
        FROM    S, SP
        WHERE   SP.qty > 100  AND
                SP.destination = S.city ).
```

There is only one way to process a TypeA nested query on a single processor. Thus, the performance of a TypeA query is fixed. The inner block must be evaluated first. Since the SELECT clause in the inner block contains an aggregate function, the evaluation of the inner block will result in a single constant rather than a list of constants. The nested predicate of the outer block then becomes a simple predicate, since the inner block can be replaced by a constant. After this, the outer block is no longer nested and can be processed completely. Thus the evaluation of inner query blocks is independent from any higher level outer blocks and all the inner blocks are evaluated only once from bottom-up.

2.2.2 TypeN Nesting

A nested query is a *TypeN* (*N* means *None*) query if the inner query block Q does not contain a join predicate which references any outer table, and the SELECT clause of Q does not contain any aggregate function. Join predicates are permitted within one query block.

For example, find out the numbers of suppliers who ship parts whose weight is greater than 50 pounds:

```
SELECT  sno
FROM    SP
WHERE   pno IN
    (   SELECT  pno
        FROM    P
        WHERE   weight > 50 ).
```

The evaluation of a TypeN nested query would be processed by first processing each inner query block Q, resulting in a list of values X which can then be substituted for the inner query block in the nested predicate. In above example, [*pno* IN Q] becomes [*pno* IN X]. The resulting query is then evaluated by either the nested-iteration method or the merge-sort method in our Oracle system, depending on the index information. Here, the evaluation of the inner block is also independent of the outer block, and the inner block is evaluated only once. The total number of the return values from the inner block has some effect on performance since these return values have to be sorted in order to remove those duplicate values. If this number is big enough, the nested iteration evaluation of the query might require extra disk I/Os.

2.2.3 TypeJ and TypeJA Nestings

A nested query is a *TypeJ* (*J* means *Join*) query when the WHERE clause of the inner query block contains at least one join predicate which references an outer table. Another condition of TypeJ nesting is that the SELECT clause of the inner query block does not contain any aggregate function. For example, select the names of parts which are stored in the place to which the parts are shipped:

```
SELECT  pname
FROM    P
WHERE   pno IN
     (  SELECT  pno
        FROM    SP
        WHERE   SP.destination = P.city ).
```

A nested query is a *TypeJA* (*JA* means *Join* and *Aggregate*) query when the WHERE clause of the inner query block contains a join predicate which references an outer table, and the SELECT clause of the inner block consists of an aggregate function over a column of an inner table. For example, find the names of parts, with the largest part number, which are shipped to the city where they are stored:

```
SELECT  pname
FROM    P
WHERE   pno =
    (   SELECT  MAX( pno )
        FROM    SP
        WHERE   SP.destination = P.city ).
```

TypeJ and TypeJA nesting are processed in most commercial systems, such as our Oracle system, by the nested-iteration method: the inner query block is processed once for each tuple of the outer table which satisfies all simple predicates on the outer table. This method has the obvious disadvantage that the inner table may have to be retrieved many times. In the examples above, the inner table SP must be retrieved once for each tuple of the outer table P, since there are no other simple predicates in the outer query block. It is this inefficiency which motivated some people to develop alternative algorithms for processing nested queries. Because of this inefficiency, we attempt experiments, reported later in this thesis, to verify the efficiency of some of these algorithms.

Query Type		*Aggregate Function in Inner Block*	*Join Predicate with Outer Table*
Single-block	Single-table	N/A	N/A
	Join query	N/A	N/A
Nested query	TypeA	Yes	No
	TypeN	No	No
	TypeJA	Yes	Yes
	TypeJ	No	Yes

Table 2.1: Summary of SQL Query Classification

2.3 Summary

Our classification for SQL queries can be summarized in Table 2.1. Note, this classification just presented is not a complete classification. We make use of this classification only for subsequent transformations and performance testing.

CHAPTER 3

The Transformation of SQL Queries

From the expressive power point of view, SQL is redundant. For a certain logical interpretation, we can usually write a query in several different, but logically equivalent, SQL forms. Thus some SQL queries can be transformed into other logically equivalent SQL queries, nonetheless this kind of transformation may be difficult to be accomplished automatically. The redundancy of SQL provides a variety of natural ways for people to conceive of, express and understand queries. However, different logically equivalent SQL queries can result in significantly different retrieval performance. Since some transformations are bidirectional, this chapter will concentrate on transformations from more structurally complex forms, such as nested query forms, into structurally simpler ones, such as join query forms.

We say that two SQL queries are *logically equivalent* if both queries produce identical answers for any tuple values of tables.

The motivation for making transformations is to determine a more efficient access plan for a proposed query. In particular, transforming a nested query into its join equivalence is desirable because the optimizers in current relational database systems that support SQL-like query languages have been designed to efficiently evaluate the join form of multiple-table queries and they resort to the nested-iteration method for evaluating most of the nested queries. The nested-iteration method is efficient only for a limited set of queries and database characteristics. In the general case, join queries are more efficient as we show later in Chapter 4.

3.1 Single-Block Query Transformations

A single-block query can always be transformed into another logically equivalent single-block SQL query by simply changing the order of the predicates (assuming there are more than one predicates). For example, a join query 'find supplier numbers and numbers of red parts which are stored in a city where a supplier stays':

```
SELECT  sno, pno
FROM    S, P
WHERE   P.color = 'red'     AND
        S.city = P.city
```

may be transformed into the logically equivalent query by

```
SELECT  sno, pno
FROM    S, P
WHERE   S.city = P.city     AND
        P.color = 'red'.
```

Changing the predicate order may effect query performance. This is why query optimizers routinely adjust the predicate order, typically to ensure projections and selections are done prior to joins. Thus, manual alternation of predicate order by programmers will have minimal impact on performance since contemporary query optimizers do a good job of this already.

3.2 Nested Query Transformations

A nested query (other than a TypeA query which may be processed only one way) can be always transformed into a series of logically equivalent single-block queries by building some intermediate tables or simply by using some join operations.

In order to introduce transformation algorithms and some examples, we assume, throughout this section, that R_o and R_i are table names for the outer table and the inner table. C_h, C_k, C_m, C_n, C_p, C_q, C_x and C_y are attribute names in the corresponding tables.

3.2.1 TypeN/TypeJ Transformations

A TypeN nested query 'find numbers of suppliers who ship parts of more than 50 pounds each':

```
SELECT  sno
FROM    SP
WHERE   pno IN
   (    SELECT  pno
        FROM    P
        WHERE   weight > 50 )
```

can be transformed into a logically equivalent query which does not contain a nested block:

```
SELECT  SP.sno
FROM    SP, P
WHERE   P.weight > 50 AND
        SP.pno = P.pno.
```

For another example, a TypeJ query 'find names of parts which are shipped to the city where they are stored':

```
SELECT  pname
FROM    P
```

```
WHERE   pno IN
    (   SELECT  pno
        FROM    SP
        WHERE   SP.destination = P.city  )
```

can be also transformed into a logically equivalent join query:

```
SELECT  P.pname
FROM    P, SP
WHERE   P.pno = SP.pno    AND
        P.city = SP.destination.
```

A lemma is given in [16] for establishing the equivalence of the TypeN or TypeJ form and the corresponding join form of a two-table query in which the operator is the set inclusion operator, IN, or other scalar comparison operators (=, ! =, <=, <, >=, >). To illustrate this lemma, consider two queries Q_1 and Q_2.

Let query Q_1 be

```
SELECT  Ck
FROM    Ro
WHERE   Ch  IN
    (   SELECT  Cm
        FROM    Ri ).
```

Algorithm TypeNJ
Begin

1. Combine the FROM clauses of all query blocks into one FROM clause.
2. AND together the whole WHERE clauses of all query blocks into one WHERE clause.
3. Replace [$R_o.C_h$ **op** (SELECT $R_i.C_m$] by a join predicate [$R_o.C_h$ **new-op** $R_i.C_m$], and AND it to the combined WHERE clause obtained on step 2. Note that if **op** is IN, the corresponding **new-op** is '='; otherwise, **new-op** is the same as **op**.
4. Retain the SELECT clause of the outermost query block.

End

Figure 3.1: Algorithm for TypeN/J Transformation

Let query Q_2 be

```
SELECT  Ro.Ck

FROM    Ro, Ri

WHERE   Ro.Ch = Ri.Cm.
```

The lemma says that Q_1 and Q_2 are equivalent and suggests a transformation algorithm ([16]) for nested TypeN or TypeJ query of depth (n-1) (here, n is the total number of block levels) to its join form. Figure 3.1 illustrates this algorithm.

By definition, the inner block of the TypeN query Q_1 can be evaluated independently of the outer block and the result of evaluating it is X, a list of values in the attribute C_m of table R_i. Q_1 is then reduced to

```
SELECT  Ck
FROM    Ro
WHERE   Ch IN X.
```

The predicate [C_h IN X] is satisfied only if X contains a constant x such that C_h = x. That is, it can be satisfied only for those tuples of R_o and R_i which have common values in the C_h and C_m columns, respectively. The join predicate [$R_o.C_h = R_i.C_m$] specifies exactly this condition. So, query Q_1 and Q_2 are logically equivalent. For a TypeJ query, the join predicate in the inner block which references an outer table is *ANDed* to other predicates. Thus, it can also be transformed to its join form by the same algorithm. For example:

```
SELECT  Ck
FROM    Ro
WHERE   Cm IN (  SELECT  Cp
                 FROM    Ri
                 WHERE   Ro.Cn = Ri.Cq OR
                         Ro.Cx = Ri.Cy )
```

could be transformed into:

```
SELECT  Ro.Ck
FROM    Ro, Ri
WHERE   Ro.Cm = Ri.Cp  AND
        ( Ro.Cn = Ri.Cq OR
          Ro.Cx = Ri.Cy ).
```

This algorithm makes a very important assumption. The result of evaluating the inner block of Q_1 is X, a list of values in the attribute C_m of table R_i. Since the list is obtained by projecting R_i over the C_m column, in general it will contain duplicate values. But if the **OP** in the nested query is IN, the effect of the simple predicate [$R_o.C_h$ IN X] is to implicitly remove any redundant values from X. However, the join predicate [$R_o.C_h = R_i.C_m$] of query Q_2 does not imply removal of duplicate values from the C_m column of R_i and the join result of Q_2 would reflect their presence. Therefore, it is assumed that when this algorithm is used for transformation and if the **OP** of the nested predicate is IN, the join query obtained after the transformation is processed by first selecting and projecting the table of the inner query block in the nested query(e.g., R_i in Q_2) and then removing duplicate values from the resulting unary table before joining it with the table of the outer query block in the nested query(e.g., R_o in Q_2). This assumption guarantees the correctness of the Lemma and the algorithm, and appears to be reasonable, since the unary table which results from projecting and selecting a table is usually much smaller than the initial table, thus the cost of joining this smaller unary table with another table is usually smaller than the cost of joining initial ones.

Also note that this algorithm can be easily extended to some nested predicates in which the **OP** is a scalar comparison operator, but a join query could be obtained from a TypeN or TypeJ nested query by that algorithm *if and only if* the **OP** of the nested predicate is IN or scalar comparison operators, which means that this algorithm does not apply when the **OP** of the nested predicate is the set noninclusion operator NOT IN. For example, the TypeJ query

```
SELECT  Ck
FROM    Ro
WHERE   Cm NOT IN
    (   SELECT  Cp
        FROM    Ri
        WHERE   Ro.Cn = Ri.Cq )
```

can not be applied by that algorithm. A nested query with the NOT IN predicate can be transformed into a query with a division operator. This is not considered in this thesis. Detailed information of this transformation can be found in [16].

3.2.2 TypeJA Transformation

A transformation algorithm ([12]) for TypeJA nested query with nesting depth of one is presented in Figure 3.2. The following example illustrates this algorithm step by step.

Let Q_3 be the TypeJA query

```
SELECT  Ck
FROM    Ro
WHERE   Ro.Cq =
    (   SELECT  AGG( Ri.Cm )
        FROM    Ri
        WHERE   Ri.Cn <= Ro.Cp ).
```

```
Algorithm TypeJA
Begin
    1. Create a temporary table TMP1 by projecting the join column of the
       outer table, and restrict it with any simple predicates applying to the
       outer table;
    2. Create another temporary table TMP2 by joining the inner table
       with TMP1. If the aggregate function is COUNT, the join here must
       be an outer join, and the inner table must be restricted and projected
       before the join is performed. If the aggregate function is COUNT(
       * ), compute the COUNT function over the join column. The join
       predicate must use the same operator as the join predicate in the
       original query(except that it must be converted to the corresponding
       outer operator in the case of COUNT), and the join predicate in the
       original query must be changed to =. In the SELECT clause, select
       the join column from the table TMP1 in the join predicate instead
       of the inner table. The GROUP BY clause will also contain columns
       from table TMP1.
    3. Join the outer table with the temporary table TMP2, according to
       the transformed version of the original query.
End
```

Figure 3.2: Algorithm for TypeJA Transformation

First, let

```
TMP1( Cp ) =
    SELECT  DISTINCT    Cp
    FROM    Ro;
```

then, another table *TMP2* is created:

```
TMP2( Cp, Cm ) =
    SELECT  TMP1.Cp, AGG( TMP3.Cm )
    FROM    TMP1, TMP3
    WHERE   TMP3.Cn <= TMP1.Cp
    GROUP BY    TMP1.Cp.
```

If *AGG* is *COUNT*, then the join for *TMP2* must be an *outer join*[3] and

```
TMP3( Cm, Cn ) =
    SELECT  Cm, Cn
    FROM    Ri;
```

otherwise, the table *TMP3* is the same as *Ri.*

Finally, a join query Q_4

[3]The *outer join* includes all values from columns participating in join, with NULLs in the opposite column if there is no match for a column value (see [8] for detail). Oracle implemented the outer join by adding a (+) after a table name in the normal join predicate.

```
SELECT  Ro.Ck
FROM    Ro, TMP2
WHERE   Ro.Cq = TMP2.Cm      AND
        Ro.Cp = TMP2.Cp
```

produces the same answer as the TypeJA query Q_3. This shows that a TypeJA query may be transformed into an equivalent join query by introducing some intermediate tables. In query Q_3, if the *AGG* in the inner query block is *COUNT(*)*, then compute the *COUNT* function over the join attribute, i.e., change *COUNT(*)* to *COUNT(Cn)*.

The algorithm in Figure 3.2 only works for the simplest TypeJA nested queries, which is nested to depth one and there is only one table in the only inner query block. In the general case, the aggregate function and the join predicate may appear at any level of nesting, and not necessarily at the same level. Thus, Algorithm TypeJA needs to be extended. The basic idea of this extension is to first remove the aggregate function from the inner query block by creating temporary tables (step 1 and step 2 in Figure 3.2), turning the TypeJA nested query into a TypeJ query, then to transform the revised TypeJ query by the TypeNJ algorithm (step 3 in Figure 3.2). In the general case, the transformation of a TypeJA query requires these two separate steps. We present a formal algorithm for the transformation of a general TypeJA query in Chapter 5.

3.3 Some Extensions

The transformation algorithms presented only consider nested predicates containing scalar comparison and set inclusion operators. But the SQL language also contains other operators such as EXISTS, NOT EXISTS, ANY and ALL. Some extensions to queries need to be implemented in order to take the advantage of these more efficient transformation algorithms.

3.3.1 EXISTS and NOT EXISTS Extensions

A nested query of the form

```
WHERE   EXISTS
    (   SELECT  selectitems
        FROM    fromitems
        WHERE   whereitems )
```

can be transformed into the nested query

```
WHERE   0 <
    (   SELECT  COUNT( selectitems )
        FROM    fromitems
        WHERE   whereitems ).
```

Similarly, a nested query of the form

```
WHERE   NOT EXISTS
    (   SELECT  selectitems
        FROM    fromitems
        WHERE   whereitems )
```

can be transformed into its equivalent

```
WHERE   0 =
    (   SELECT  COUNT( selectitems )
        FROM    fromitems
        WHERE   whereitems ).
```

These transformations may result in a TypeA or TypeJA nesting depending upon whether or not the inner query block has any join predicate with an outer table, and these transformations are bidirectional.

3.3.2 ANY and ALL Extensions

A nested predicate of the form

```
< ANY  (    SELECT  selectitem
            FROM    fromitems
            WHERE   whereitems )
```

can be transformed into the logically equivalent form

```
< ( SELECT  MAX( selectitem )
    FROM    fromitems
    WHERE   whereitems ).
```

The same transformation is performed when the operator is $<=$ *ANY*.

Similarly,

```
< ALL  (    SELECT  selectitem
            FROM    fromitems
            WHERE   whereitems )
```

can be transformed into the logically equivalent nested predicate

```
< ( SELECT  MIN( selectitem )
    FROM    fromitems
    WHERE   whereitems ),
```

and the same transformation is performed when the operator is $<=$ *ALL*. If the comparison operator is $>$ or $>=$, the transformation is the reverse:

```
> ANY  (    SELECT  selectitem
```

can be changed to

```
> ( SELECT  MIN( selectitem )
```

and

```
> ALL  (   SELECT  selectitem
```

can be changed to

```
> ( SELECT  MAX( selectitem ).
```

More simply, an operator of the form *=ANY* is transformed into IN, and an operator of the form *!=ANY* is transformed into NOT IN. But the set noninclusion operator is not considered in our performance testing, so the transformation of *!=ANY* is not considered further.

3.4 Cost Models for Transformed Query Processing

In order to convince ourselves that the processing performance will be better after these transformations, we need to set up cost models in order to compare the resulting candidate queries.

Let us assume:

R_o: an outer table;

R_i: an inner table;

R_t: the temporary table *t* obtained by intermediate processing on R_i, *t* is the table name;

P_k: the size in pages of table R_k;

N_k: the number of tuples in table R_k;

f_k: the fraction of the tuples of table R_k that satisfy all simple predicates on R_k;

B: the size in pages of available main memory cache.

(Note, k represents a table name, thus $k = o,\ i\ or\ t$.)

Furthermore, the nested query has the simplest form: one outer block and one inner block, the *FROM* clause of each block contains only one table. No indexes are available for query processing (i.e., tables are sequentially scanned). The nested-iteration method is used for the evaluation of TypeN, TypeJ and TypeJA queries, and a $(B-1)$-way multiway merge-sort method is used for all the join operations, which requires $(2 \cdot P_k \cdot \log_{B-1} P_k)$ page I/Os to sort the table R_k.

For a TypeN query, the inner query block is evaluated first, thus the temporary table R_t is a unary table with smaller size than the initial table. For a TypeJ query, the temporary table R_t is obtained by selecting and projecting on table R_i first according to the simple predicates in the inner query block. This might reduce the size of R_t. If table R_t can not fit into the $(B-1)$ page main memory cache, then R_t has to be fetched once for each tuple of R_o that satisfies all other simple predicates on R_o, as many as $(f_o \cdot N_o)$ times. The cost is up to

$$(P_i + P_t) + P_o + f_o \cdot N_o \cdot P_t \quad (page\ \ I/Os)$$

where the first two terms are the cost of generating R_t since table R_t has to be written out on disk. In contrast, the total cost for the equivalent join query by the merge-sort method is

$$(P_i + P_t + 2 \cdot P_t \cdot \log_{B-1} P_t) + (P_o + P_{t1} + 2 \cdot P_{t1} \cdot \log_{B-1} P_{t1}) + (P_t + P_{t1}) \quad (page \ \ I/Os)$$

where the first three terms are the cost of generating R_t from R_i and removing duplicates from R_t by sorting; the next three terms are the cost of selecting and projecting R_o into R_{t1} and sorting it, thus $P_{t1} = f_o \cdot P_o$; and the last two terms are the cost of merge joining R_t and R_{t1} after sorting both of them.

For example, for a TypeN or TypeJ query, if we have six pages of main memory cache ($B = 6$), the size of the inner and outer tables is 100 pages ($P_i = P_o = 100$), there are 500 tuples in table R_o which satisfies all other simple predicates ($f_o \cdot N_o = 500$), and the temporary table R_t has 20 pages ($P_t = 20$), according to our cost model, the nested-iteration method may cost 10,220 page I/Os. If the table R_{t1} has 50 pages ($P_{t1} = 50$) and a five-way merge-sort is used to sort R_t and R_{t1}, the join query costs total 658 page I/Os. For another example, if we have six pages of main memory cache ($B = 6$), the inner and outer table size is 50 pages ($P_i = P_o = 50$), no simple predicate for the outer table ($f_o = 1$) and there are 500 tuples in the outer talbe ($N_o = 500$), and the table R_t has 20 pages ($P_t = 20$), the nested-iteration method costs 10,120 page I/Os. If a five-way merge-sort join is used for the join query, the join processing needs 558 page I/Os. Thus, transformed join queries are more efficient than nested queries. More detailed theoretical analysis results can be seen in [16].

The cost for a TypeJA query evaluated by the nested-iteration method is

$$P_o + f_o \cdot N_o \cdot P_i \quad (page \ \ I/Os).$$

The total cost of using the TypeJA transformation algorithm in Figure 3.2 will consist of three major sub-costs:

1. The projection and selection on the outer table R_o, resulting in a temporary table *TMP1*;
2. The creation of temporary table *TMP2* by projecting and selecting the inner table R_i, joining this with temporary table *TMP1*, and performing a GROUP BY operation on the result;
3. Joining the temporary table *TMP2* with the outer table R_o.

Thus, the total cost of transforming a simplest TypeJA nested query will depend on the type of join used to create temporary tables (since a join might have to be an outer join), and also depend on the type of join used between the outer table R_o and the temporary table *TMP2*. If the merge-sort join is used for all of the join operations, the normal join and the outer join will have the same cost.

The cost of creating the table *TMP1*, e.g., projecting and selecting R_o, then sorting it, is:

$$P_o + P_{TMP1} + 2 \cdot P_{TMP1} \cdot \log_{B-1} P_{TMP1} \quad (page \;\; I/Os)$$

The cost of creating the temporary table *TMP2* including the GROUP BY operation is:

$$P_i + P_{TMP3} + 2 \cdot P_{TMP3} \cdot \log_{B-1} P_{TMP3} + P_{TMP1} + P_{TMP3} + 2 \cdot P_{TMP4} + P_{TMP2} \quad (page \;\; I/Os)$$

where table TMP4 is the result of join before the GROUP BY operation. The first three terms above are the cost of creating a sorted *TMP3* from table R_i. The next two terms are the cost of merge join *TMP1* and *TMP3*. Then, perform the GROUP BY operation on *TMP4* and write out table *TMP2* to disk. The cost of performing the final join between R$_o$ and *TMP2* is:

$$2 \cdot P_o \cdot \log_{B-1} P_o + P_o + P_{TMP2} \quad (page \;\; I/Os)$$

Thus, the total cost of the algorithm is the summary of these three parts as given above.

The cost model for TypeJA transformation can be compared to the nested-iteration method in the following example. The TypeJA query is query Q3 above, the aggregate function is *COUNT*, but in the inner query block, the join is an equal join instead of with '<=' operator. For example, the outer table has 50 pages ($P_o = 50$), the inner table has 30 pages ($P_i = 30$), the corresponding temporary table R_{TMP1}, R_{TMP2}, R_{TMP3} and R_{TMP4} have 7 pages, 5 pages, 10 pages and 8 pages respectively ($P_{TMP1} = 7, P_{TMP2} = 5, P_{TMP3} = 10, P_{TMP4} = 8$), the main memory cache is 6 pages ($B = 6$), there are 100 tuples which satisfy the simple predicates on the outer table ($f_o \cdot N_o = 100$). The nested-iteration method of processing the query costs 50 + 100 * 30 = 3,050 page I/Os. The transformation method using Algorithm TypeJA and two merge-sort joins cost about 470 page I/Os. This example shows that the nested-iteration method is less efficient under the assumptions in the example. The theoretical analysis of Algorithm TypeJA is very complex, and a detailed analysis is presented in [12].

From the analyses above, we can determine that, generally speaking, a theoretical analysis for an algorithm is clear, correct, but has some limitations in the sense that:

1. All of the theoretical analyses are based on simplified models, like simplified queries and simplified memory situations. Certain statistical properties, e.g. uniform distribution and independence of attribute values, are commonly assumed. This can only reflect a part of the performance information, sometimes, even a rather small part. For more general, more practical situations, this kind of theoretical analysis is difficult to do and to compare, thus the conclusions obtained based on those assumptions are suspect.

2. Most theoretical analyses are based entirely on a small number of processing strategies. Index information can play a very important role on query performance and the strategy chosed by a query optimizer sometimes is related to the available indexes. But adding index information to the analysis makes the analysis more complicated and even more difficult.

3. Frequently the number of secondary storage accesses is the sole, or at least dominating, cost measure, as in the analyses above. This assumes that disk I/O is still the bottleneck of SQL query processing. But today's transaction processing systems often have large database buffers and are CPU-bound rather than I/O-bound. Multitasking environments, buffer management, concurrency control, communication cost and operating system overhead are frequently neglected by the traditional analysis method, even though they have a major performance impact sometimes.

Therefore, a performance testing on an actual RDBMS can make the conclusions more convincing.

CHAPTER 4

Performance Tests with Wisconsin Benchmark

A *benchmark* is a point of reference from which measurements of any sort may be made[4]. The need for benchmarks arises whenever there are different products claiming or providing similar functionality. For example, in industry there is a trend to decentralize data management which has acted as a catalyst to the development of relational distributed database management systems. Some of these RDBMSs include Distributed INGRES by Ingres Corporation, SDD-1 developed at the Computer Corporation of America and R* developed by IBM. Because these systems offer similar functionalities, a need to perform a comparative evaluation of their performance becomes necessary. A vendor could also use benchmarks to stress test a system under development. Another use for a vendor is in establishing a particular rating for a

[4]See *Webster's Third New International Dictionary*, 1981.

system. Finally, a user can use a benchmark to compare several systems before purchasing one. Benchmarks are also useful to logical and physical database designers because benchmarks help the designers identify costs and highlight problems users may face given certain design decisions.

A good benchmark for database systems should have the following basic characteristics ([5]):

1. **Single-user/Multi-user modes**: A good benchmark should come in both the single-user and multi-user modes. The single-user benchmark should provide best case expected performance of a system, while the multi-user benchmark should test the system under normal operating conditions.
2. **Scalatility and Portability**: A good benchmark should be scalable so that systems of various sizes can be meaningfully compared. It should also be easily portable across platforms.
3. **Ease of Implementation**: The benchmark should be easy to implement. A benchmark does not have to require many person-months to set up.
4. **Database Structure**: Most existing benchmarks use one database. This suggests a centralized system by default. If a benchmark is to be run on systems that are not centralized, then the structure of the benchmark, including the structure of the database(s) used, should clearly reflect that fact.
5. **System Workload**: The workloads faced by the systems under test should be modeled as closely as possible.

6. **Performance Metrics**: The performance metrics measured by the benchmarks should be clearly stated and defined.

4.1 Benchmark Methodology for Database Performance Testing

Managing a database requires a complex system composed of hardware, software, and data components. A benchmark methodology for database systems must consider a wide variety of system variables in order to fully evaluate performance. Each variable must be isolated as much as possible to allow the effects of that variable, and only that variable, to be evaluated.

The benchmark methodology for database systems consists of three stages ([30]):

1. **Benchmark Design**:

   Establishing the environment of the database systems to be tested, and developing the actual tests to be performed, which includes setting up the system environment for the benchmark; designing the system configuration, test data, workload, and variables of the benchmark studies.

2. **Benchmark Execution**:

   Performing the benchmark testing and collecting the performance data.

3. **Benchmark Analysis**:

   Analyzing the performance results on individual database systems and, if more than one system is benchmarked, comparing performance across several systems. The performance experiment of this thesis will be run on one relational database system for different, but logically equivalent, SQL queries.

4.2 Benchmark Design

In the past few years we have seen in the literature a number of proposals for benchmarks to be used in measuring the performance of database management and transaction processing systems. The TP1 benchmark ([1]) and the Wisconsin benchmark ([3], [5], and [4]) have been used to benchmark several systems. Other benchmarks have also been proposed. It appears as though both the TP1 and the Wisconsin benchmark have the potential of becoming *de facto* standard benchmarks, in their respective areas, to be used in a variety of ways.

Whereas TP1 is oriented towards transaction processing, the Wisconsin benchmark was conceived for the purpose of measuring the performance of relational database systems. It consists of two parts: a single user benchmark in which a suite of approximately 30 different queries are used to obtain response time measurements in stand alone mode (described in [3]) and a multi-user benchmark in which several queries of varying complexity are used to determine the response time and throughput behavior under a variety of conditions (one version of the multi-user benchmark is described in [5] and the second version in [4]).

In the Wisconsin benchmark, the test database consists of a number of relations of varying sizes. The relations are generated according to statistical distributions and do not model any real-world data. Users of the benchmark can modify the database generator routines to adapt the database characteristics so that they are more representative of their application.

4.2.1 Wisconsin Benchmark Database

Our benchmark experiment will be run on our Oracle system using the Wisconsin Benchmark database as the test database. The Wisconsin benchmark was one of the first attempts at formalizing experimental performance evaluation of relational database systems. It was originally conceived as an experiment in benchmarking methodology. The benchmark focuses on measuring the performance of access methods and query optimization in a relational database system. Since the purpose of this thesis work is to study the performance information between logically equivalent SQL queries for a database, and to test the performance difference in different index situations, we chose to use the Wisconsin benchmark database.

The original test database consists of 3 relations, with identical attributes but different cardinalities, one with 1,000 tuples and the other two with 10,000 tuples. Each relation has 16 attributes, 13 two-byte integer attributes and three fixed-length string attributes with 52 bytes for each attribute. This results in the tuple width of 182 bytes in total. For a table in a benchmark database, this number can avoid giving any system an advantage through some fortuitous alignment of tuples in pages. Each of the three string attributes has three distinguishing characters occurring in positions 1, 27 and 52. These distinguishing characters allow for 26^3 unique strings,

Name	*Type*	*Range*	*Order*	*Comment*
unique1	int	0 - 9999	random	candidate key
unique2	int	0 - 9999	random	declared key
two	int	0 - 1	rotating	0, 1, 0, 1, ..
four	int	0 - 3	rotating	0,1,2,3,0,1, ..
ten	int	0 - 9	rotating	0,1,..,9,0,1, ..
twenty	int	0 - 19	rotating	0,1,..,19,0,..
hundred	int	0 - 99	rotating	0,1,..,99,0,..
thousand	int	0 - 999	random	
twothous	int	0 - 1999	random	
fivethous	int	0 - 4999	random	
tenthous	int	0 - 9999	random	candidate key
odd100	int	1 - 99(50)	rotating	1,3,5,..,99,1,
even100	int	0 - 98(50)	rotating	0,2,4,..,98,0,
stringu1	char	a..a..a - v..v..t	random	candidate key
stringu2	char	a..a..a - v..v..t	rotating	candidate key
string4	char	a..a..a - v..v..v	rotating	

Table 4.1: Description of the Attributes in Table TenKOne of Wisconsin Benchmark

thus enough for the 10,000 tuple table. The remainder of the positions contain the same padding character. See Table 4.1 for a detailed description of each attribute in the table *TenKOne*.

The smaller table *OneK* has the same attributes as the table *TenKOne*, with identical ranges and cardinalities except where the number of tuples in the table precludes some attributes from having all of the integer values within the specified range.

In order to test the effects of indexes, we created three indexes on each table: a clustered index on ***unique2***, a nonclustered, but unique index on ***unique1*** and another nonclustered, but nonunique index on ***hundred.***

Some system designers and users of database systems who implemented the benchmark have criticized the original design on numerous points ([2]). Among the most common criticisms are those about the structure and size of the database, the tuple length, data type structure of the strings and distributions of attribute values, the difficulty in scaling the benchmark to various applications, the restricted and unrealistic set of test queries and the fact that the single user mode is not representative of a system's performance in an actual application.

4.2.2 Index Information and Query Performance

Conceptually speaking, an *index* is a binary relation that associates certain attribute value(s) with references to relation elements (tuples), usually called tuple identifiers.

We consider three kinds of index:

1. cluster: the tuples with the same value on the cluster column are stored together physically, there can be only one clustered index for any table;
2. noncluster/unique: there is exactly one tuple in the table for each index key;
3. noncluster/nonunique: there is one or more tuples in the table for each index key.

An index can be composed of one or more attributes. An index with more than one attribute is called a *concatenated index.*

A relational system does not automatically build indexes, rather they must be created by authorized users such as database administrators. Index selection is not trivial, since an index designer must balance the advantages of indexes for data retrieval versus their disadvantages in maintenance costs (incurred for database inserts, deletes, and updates) and database space utilization. An index always plays an important role in the efficiency of certain searching times since some searching can be accomplished by simply scanning the index itself and some other searching can be accomplished by the direct access of data blocks through the index. Nonetheless, a poor choice of index designs can result in poor system performance, far below what the system would do if a better set of indexes were available. Furthermore, the existence of certain indexes, although they improve the performance of some statements, may reduce the performance of other statements, since the indexes must be modified when tables are updated.

Thus, indexes have the following advantages and disadvantages.

Advantages

- An index may speed up direct access based on a given value for the indexed column or column combination. Without the index, a sequential scan would be required.
- Indexes speed up sequential access based on the indexed column or column

combination. Without the index, a sort would be required.

Disadvantages

- Indexes require space on disk. The space devoted to indexes can easily exceed that taken up by the data itself in a heavily indexed database.
- Although an index can speed up retrieval operations, it will, at the same time, slow update operations. Any update on the indexed column or column combination will require an accompanying update operation on the index.

4.2.3 Query Design

The purpose of this work is to determine the performance relationship between logically equivalent SQL queries in situations with different index information available. Thus, the original Wisconsin benchmark queries are not suitable for this work, a group of new queries needs to be designed. According to our classification of SQL queries, the query design is trying to take the advantage of the Wisconsin benchmark database in order to make use of different indexes and, at the same time, control the query selectivity, e.g., the size of the query answer, as well. Thus, we follow three basic principles for query design:

1. Each category in our query classification should be tested;
2. Each query should be tested with different index information;
3. The impact of the number of tuples returned should be considered.

Our classification of SQL queries yielded seven groups of test queries. For each group, there are several logically equivalent queries derived from the transformation algorithms. Group1 and Group2 are used to test the performance for single-block queries in different index situations. Group3, Group4 and Group5 are to used test the performance for single-level nested queries. Group6 and Group7 are used to test the performance for some extensions. Only the nested queries of depth one with the only inner block were tested. The SQL queries that are introduced in this chapter are also summarized in Appendix A.

According to the definition of the benchmark tables, attributes *tenthous* has no index, *unique1* has a nonclustered unique index. Each of them is a key for that table since each value of each of those attributes can uniquely identify a tuple in that table. The value range and value distribution is identical for these two attributes. Attributes *tenthous, unique1* and *hundred* were used in test queries to determine the index impact.

When we designed these test queries, the number of return tuples, i.e., the size of the query answer, was a very important factor which we considered. In different index situations, we tried to determine what impact the size of the answer had on retrieval time. Again, the key attributes *tenthous* and *unique1* were used to control the query answer size.

Another assumption for the TypeNJ transformation algorithm is that if the **OP** is the set inclusion operator IN, the corresponding join query must remove duplicates before the actual join processing. In our tests, we always *SELECT*ed from a key attribute in the inner table if the **OP** is IN. Thus, the intermediate table after processing the inner block would not contain any duplicates. This simplified the duplicate removing processing, and the performance conclusions are still valid since in some actual situations, duplicates do exist. Thus the intermediate table which results from projecting and selecting the inner table is usually much smaller than the initial one. The cost of joining this reduced table is, therefore, usually smaller than the cost of joining the initial table. Thus, this simplified testing demonstrates the worse case because the intermediate table is not reduced in size.

4.3 Benchmark Execution and Analysis

All of the performance tests were run on SunOS Release 4.1.1 in single-user mode during weekends or late nights. The host machine for the Oracle database[5] is also a file server, so for each test, we chose the shortest timing after at least 10 executions of the same query. Since this test was in single-user mode, only the measurements for response time were reported. The SQL queries were executed from a Pro*C program. Timing data was obtained by the use of system calls. For example, in the UNIX environment, we made use of the *gettimeofday()* system call before and after a query to determine the elapsed time for response time. A sample Pro*C program used in the testing appears in Appendix B.

In our Oracle system, an integer is 4 bytes, thus the tuple width in our tests is

[5]Oracle RDBMS, Version 6.1.

208 bytes. The database data block size on our Oracle system is 2k, and the main memory cache has 200 blocks. For our tables, table *OneK* requires 178 blocks and *TenKOne/TenKTwo* requires 2119 blocks respectively. Thus, table *OneK* can fit into main memory cache, thus occasional improving the performance significantly, especially for a nested iteration algorithm if *OneK* is the table used in the inner loop.

In SQL*Plus[6], there is a performance diagnostic tool which can be used to query the access plan chosed by the optimizer. All the access plans shown in later sections were obtained by using this tool. Detailed information for this diagnostic tool can be obtained in [22].

4.3.1 Single-Table Queries

The queries used for testing are

1.1:

```
SELECT  even100
FROM    TABLE
WHERE   odd100 < 100    AND
        two < 2         AND
        unique1 < value;
```

[6]SQL*Plus is an interactive command language for working with an Oracle database. Detailed information can be obtained from [21].

```
1.2:

        SELECT  even100
        FROM    TABLE
        WHERE   unique1 < value AND
                odd100 < 100    AND
                two < 2.
```

The SQL query 1.1 and 1.2 above are logically equivalent since one may be obtained from the other simply by switching the predicate order. Here, *TABLE* could be *OneK, TenKOne* or *TenKTwo*, *value* is an integer value to restrict the number of return tuples. Since *odd100* is an odd number from 1 to 99 and *two* can only be 0 or 1, these two predicates are always true. Therefore, the *value* is the number of return tuples. According to the query planner on our system, for query 1.1, no matter what order the predicates are in, the execution plan, when *TABLE* is *TenKOne*, is always:

```
        TABLE ACCESS BY ROWID TenKOne
          INDEX RANGE SCAN tenk1unik
```

where, *tenk1unik* is the index name on *TenKOne.unique1.* Thus the query plan is not effected by the order of the predicates, and if there is an index available, the optimizer will use the index instead of doing a full table scan to execute that query. This means that the Oracle optimizer will optimize the query based on the index information instead of the query syntax like the order of the predicates.

Our test data (see Table 4.2 and Table 4.3) shows that on Oracle, both queries for the same table require approximately the same execution time, no matter the size of the table, as long as both queries have the same index available. Thus, the query

		Number of Return Tuples						
Query	*Index*	*10*	*100*	*200*	*300*	*400*	*500*	*1000*
1.1	Yes	0.13	0.79	1.50	2.24	2.95	3.61	7.16
1.2	Yes	0.13	0.78	1.48	2.25	2.97	3.61	7.13
1.2	No	0.50	1.11	1.77	2.43	3.14	3.73	7.11

Table 4.2: Timing Data(Sec.) for Single-Table Queries with Table OneK

		Number of Return Tuples							
Query	*Index*	*100*	*1000*	*2000*	*3000*	*4000*	*5000*	*8000*	*10000*
1.1	Yes	0.79	8.33	16.69	24.99	32.56	40.99	65.35	82.16
1.2	Yes	0.80	8.38	16.57	25.13	32.76	40.77	65.39	81.93
1.2	No	5.49	11.24	17.67	23.82	30.55	36.55	55.95	68.13

Table 4.3: Timing Data(Sec.) for Single-Table Queries with Table TenKOne

performance cannot be changed simply by changing the predicate order in a query. This test indicates that the Oracle optimizer can always find a good way to evaluate single-table queries.

But, if we run the same query with and without an index, the situation is much different. However, an index does not always lead to greater efficiency. For a smaller table like *OneK*, an index is always helpful. This is due to the fact that both the table data and the index data can fit into the main memory cache. In this case, the main memory cache is 200 blocks, *OneK* requires 178 blocks, and the index on *unique1* requires only 11 blocks. Even when the entire table has to be accessed, the overhead of accessing the index hardly has any effect on query performance. But for a larger table like *TenKOne*, a query executed with an index can be less efficient than the full

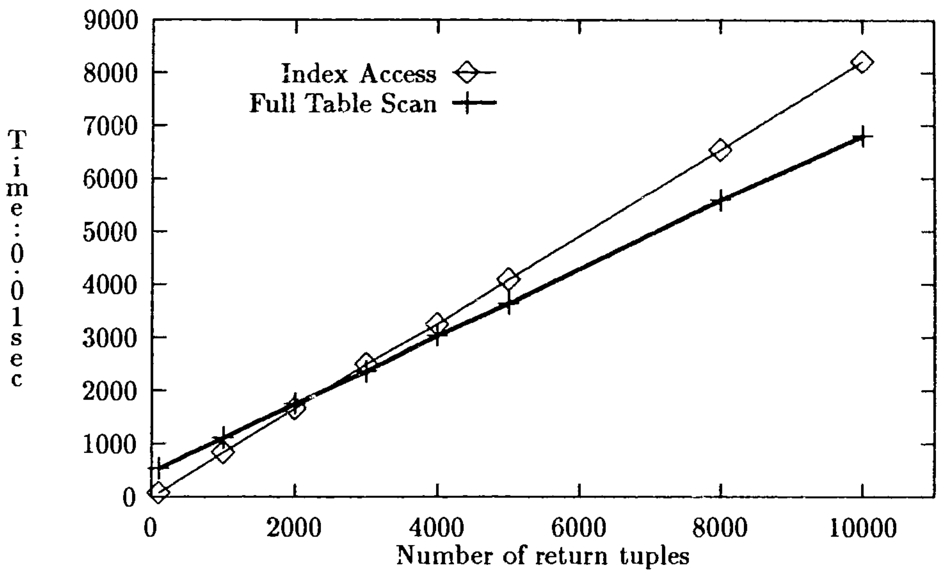

Figure 4.1: Single-Table Query 1.1 for Table TenKOne

table scan sometimes. This can be seen from Figure 4.1. The *TenKOne* table data occupies 2119 blocks, and the index on *unique1* requires 99 blocks. Searching by index might cause some data blocks to be read more than once, and if that required data block is not in the cache at that moment, more disk I/Os will result. In this case, an index can only be more efficient if the selectivity of that query is relatively high, which means the number of return tuples of that query is not too large. From our tests with Oracle, this number might be around 25% to 30% of the table size. If the proportion of return tuples is more than that, an index can make the performance worse. When the selectivity is low, large parts of the index and the table will be accessed. In many cases it will take less time to scan the entire table than to access it using the index. When accessing the table via the index, much time will be spent moving the disk arm between the index data and the table data. In the extreme situation (for example, when a query returns every single tuple in the table *TenKOne*), the index access becomes complete overhead. A whole table scan of *TenKOne* can be 22% faster in this case. This difference can be even larger for huge tables.

4.3.2 Join Queries

Two different join queries were employed.

```
2.1:

        SELECT  TenKOne.even100
        FROM    TenKTwo, TenKOne
        WHERE   TenKTwo.value2 < value3          AND
                TenKTwo.value2 = TenKOne.value1
2.2:

        SELECT  TenKOne.even100
        FROM    OneK, TenKOne, TenKTwo
        WHERE   OneK.hundred < value             AND
                TenKOne.unique2 < value          AND
                TenKTwo.hundred < value          AND
                OneK.hundred = TenKOne.unique1   AND
                OneK.hundred = TenKTwo.hundred
```

We notice by consulting the Oracle planner that the actual access plans of these two queries are independent of the predicate order in the queries, which means the timing data should be the same if the predicate order is the only difference between two queries. In Table 4.4, query 2.1 was obtained by *value1 = tenthous, value2 = unique1, value3* goes from 100 to 10000. Query 2.1' was obtained by putting the join predicate before the selection predicate in query 2.1 (similarly for query 2.2' in Table 4.5). From

	Number of Return Tuples						
Query	*100*	*1000*	*2000*	*4000*	*6000*	*8000*	*10000*
2.1	14.11	20.19	26.86	40.10	52.56	66.15	78.69
2.1'	14.31	20.53	27.54	38.54	53.40	64.57	79.55

Table 4.4: Timing Data(Sec.) for Two-Way Join Query 2.1

	Number of Return Tuples						
Query	*1000*	*10000*	*20000*	*40000*	*60000*	*80000*	*100000*
2.2	9.58	95.56	191.31	387.20	561.92	748.60	938.94
2.2'	9.59	95.61	190.80	389.42	563.63	750.94	934.00

Table 4.5: Timing Data(Sec.) for Three-Way Join Query 2.2

those two tables, we can see that no significant performance difference may be derived by switching the predicate orders in join queries.

We tested the following index situations for query 2.1: neither of the two tables has an index, one of them has an index on *unique1*, and both of them have indexes on *unique1*. The value for *value3* ranges from 100 to 10,000. Here again, we always

Table	*Table*	*Number of Return Tuples*						
TenKOne	*TenKTwo*	*100*	*1000*	*2000*	*4000*	*6000*	*8000*	*10000*
		20.94	27.20	34.52	48.78	65.99	82.90	99.51
index	index	0.88	9.15	18.52	38.15	55.33	74.32	92.70
index		6.35	15.17	24.79	42.40	61.29	80.39	99.42
	index	14.11	20.19	26.86	40.10	52.56	66.15	78.69

Table 4.6: Timing Data(Sec.) for Join Query 2.1 with/without Indexes

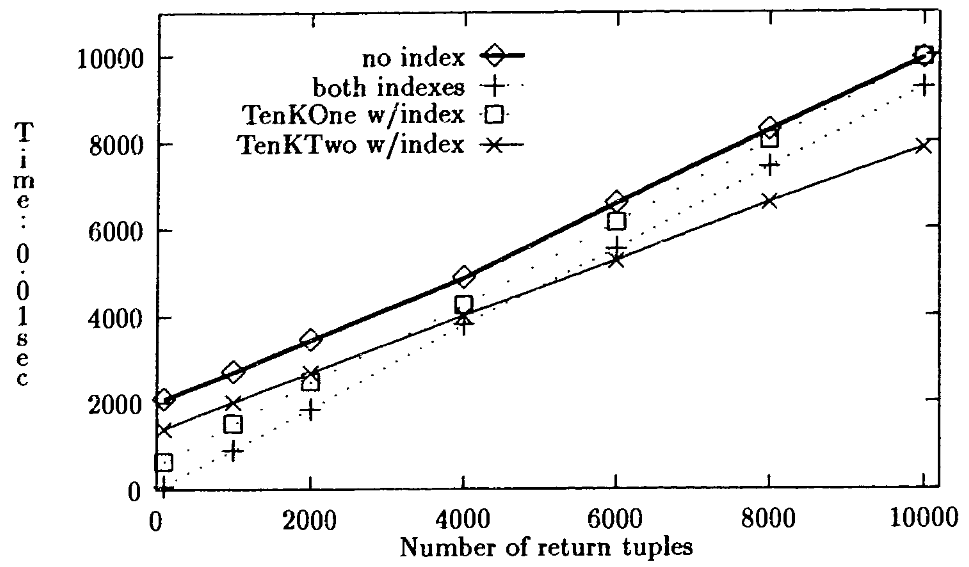

Figure 4.2: Join Query 2.1 with Different Indexes

choose a key attribute for *value1* and *value2*, so that, for every value of *value1* in table *TenKOne* there is exactly one tuple for *value2* in *TenKTwo*. Therefore, *value3* is the number of return tuples for that query. From Table 4.6 and Figure 4.2, we can see that, in this case, using an index does significantly improve performance. This is because the join attributes are keys for the two tables, and only one tuple is accessed from each table. But for a given query, different index information will cause the optimizer to use totally different access plans, and those access plans would play a major role for query performance.

In query 2.1, if none of the concerned attributes in the WHERE clause has an index, the *merge-sort method* will be used for the evaluation. The sorting for each table is by the full table scan since no index is available. Thus, the timing data for this case is higher than for all of the other cases (See Figure 4.2).

If both *value1* and *value2* have an index in query 2.1, then the *nested-loop method* will be used. The access plan is

```
NESTED LOOPS
  INDEX RANGE SCAN tenk2unik
  TABLE ACCESS BY ROWID TenKOne
    INDEX UNIQUE SCAN tenk1unik
```

In this situation, *tenk2unik* and *tenk1unik* are the indexes for table *TenKTwo* and *TenKOne*. Since the join attribute and the select attribute for *TenKTwo* is the index attribute, i.e., *value2*, the valid values are obtained by scanning the index only instead of accessing the table. For each value of that attribute which satisfies the selection condition obtained by using the index range scan method, use the index on *TenKOne.value1* to produce the join result. Having both tables indexed is best if less than half the tuples in both tables are accessed for that join operation.

If only *TenKOne* has an index, the access plan is

```
NESTED LOOPS
  TABLE ACCESS FULL TenKTwo
  TABLE ACCESS BY ROWID TenKOne
    INDEX UNIQUE SCAN tenk1unik
```

The index is used to access *TenKOne*, but the driving table *TenKTwo* has to be full table scanned. This makes this case constantly worse than both-index-case due to the difference between a full table scan and an index access of the driving table. But this situation is still better than no-index-case most of the time.

If only *TenKTwo* has an index, the nested-loop method is still used. The access plan becomes

```
NESTED LOOPS
  TABLE ACCESS FULL TenKOne
  INDEX UNIQUE SCAN tenk2unik
```

But this time, the driving table is *TenKOne* instead of *TenKTwo*. The driving table is full table scanned. For each tuple in *TenKOne*, the index on *TenKTwo* is used to check the corresponding value for the join attribute. This means that the index on *TenKTwo* has to be accessed as many times as the number of tuples in *TenKOne*. Thus, if only a small number of tuples in *TenKOne* is in the join output, accessing each single tuple of *TenKOne* and going through the whole index *tenk2unik* on *TenKTwo* would be an overhead for that join. This is why TenKTwo-index-case is the worst among all the index cases for small retrievals. But if more than half of the *TenKTwo* satisfy the selection condition, which means more than half of the tuples in *TenKOne* will be in the join output, the full table scan is the best way. Thus, this case is the most efficient one for large retrievals.

From the four cases described above, we can observe that indexes can always make join queries more efficient. The query optimizer can recognize the available indexes and always take the indexed table as the inner loop table in a nested-loop method, thus making the join processing more efficient sometimes if there are some indexes available. But taking the indexed table as the inner loop table in a nested loop join method is not always efficient as in the case of next section.

4.3.3 TypeN Queries

A TypeN query can be transformed into a logically equivalent join query. The TypeN query and the join query used for the test are

```
3.1:
        SELECT  even100
        FROM    TenKOne
        WHERE   value1 IN
             (  SELECT  value2
                FROM    TenKTwo
                WHERE   value2 < value3  ),
3.2:
        SELECT  TenKOne.even100
        FROM    TenKTwo, TenKOne
        WHERE   TenKTwo.value2 < value3  AND
                TenKTwo.value2 = TenKOne.value1.
```

In order to understand the role an index plays in a TypeN query, four cases (outer block with and without index and inner block with and without index) are considered. In our testing, only a nonclustered unique index is used. The value range for *value3* is from 100 to 10,000.

The inner query block in a TypeN nesting may be evaluated first. By consulting the Oracle optimizer, we can see that, for a TypeN query, if there is an index in the outer block, then the *nested-iteration* method is always used. This method sorts the return values from the inner block first, then for each value, uses the index in the outer block to access the outer table in order to select the valid tuples. Whether or not the inner block has an index for this query only determines whether the inner table is full-table-scanned or accessed through the index. If no index is available for the outer block, the *merge-sort* method is used, no matter whether the inner block has an index or not. First the return values from the inner block are sorted, then the outer table *TenKOne* is full-table-scanned in order to sort it. After these steps, a join operation is executed. Equivalent join queries have exactly the same access plans as in the previous section.

From Table 4.7. we observe that in all the cases, the join query is never less efficient than the corresponding TypeN query. But the transformation from TypeN to join achieves more benefit when the inner query block, thus the outer block too, returns a large number of values. This can also be deduced from Figure 4.3 when the outer block does not have any index, but the inner block may have a nonclustered unique index.

If the number of return tuples of a query is more than 4,000, there is a dramatic performance increase for some index cases (see Table 4.7). This is because in those

Query	*Outer Table*	*Inner Table*	*Number of Return Tuples*						
			100	*1000*	*2000*	*4000*	*6000*	*8000*	*10000*
TypeN		Index	16.23	22.13	29.82	44.92	64.14	82.75	98.51
Join			14.11	20.19	26.86	40.10	52.56	66.15	78.69
TypeN			20.98	27.42	34.40	49.58	76.50	90.96	104.52
Join			20.94	27.20	34.52	48.78	65.99	82.90	99.51
TypeN		Index	0.94	9.72	19.56	38.78	59.66	80.17	99.19
Join	Index		0.88	9.15	18.52	38.15	55.33	74.32	92.70
TypeN			6.99	15.34	25.12	42.98	65.84	85.76	104.55
Join			6.35	15.17	24.79	42.40	61.29	80.39	99.42

Table 4.7: Timing Data(Sec.) for TypeN/Join Queries (3.1/3.2)

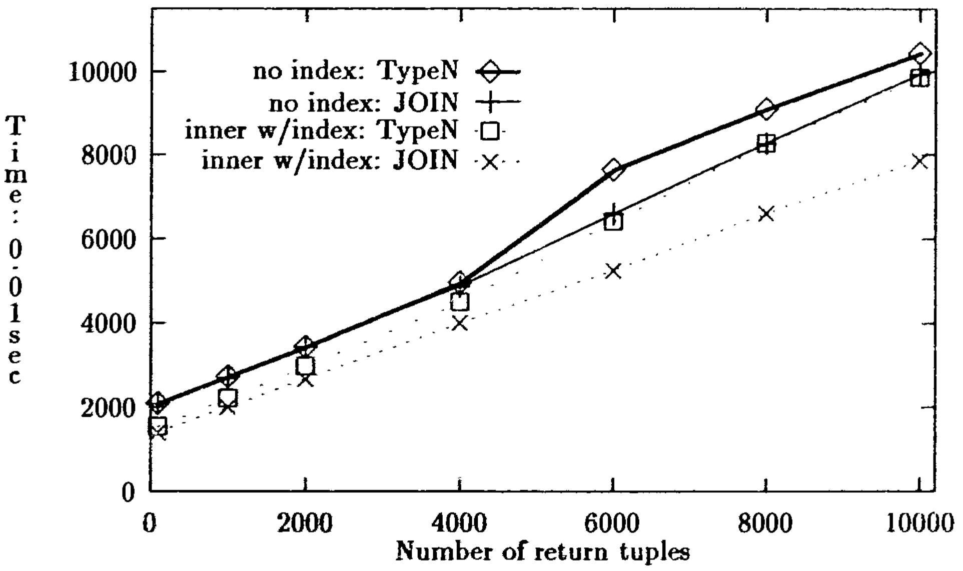

Figure 4.3: TypeN Query 3.1 with Merge-Sort Method and Join Query 3.2

cases, the returning values from the inner block are always sorted for the IN operator in the WHERE clause of the outer query block. When the number of these intermediate values returned are large enough, the sorting will take more time. This may be due to the fact that this sorting process causes some disk I/Os (The size of main memory cache available for sorting in Oracle is unknown for this case). This accounts for why, even in no-index situation in which both the TypeN and Join queries use the same merge-sort method, TypeN is still slower than join. We can see this behaviour from Figure 4.3 as well.

All the above tests are based on two tables neither of which can fit into the main memory cache. In order to find out what will happen if at least one of them fits into the cache, we composed another pair of queries.

```
3.3:

        SELECT  even100
        FROM    TenKOne
        WHERE   hundred  IN
             (  SELECT  tenthous
                FROM    OneK
                WHERE   tenthous < value3 )
3.4:

        SELECT  TenKOne.even100
        FROM    OneK, TenKOne
        WHERE   OneK.tenthous < value3    AND
                OneK.tenthous = TenKOne.hundred.
```

In these two queries, *OneK* can fit into the main memory cache. Since the number of return tuples should be *100*value3*, the value for *value3* ranges from 1 to 100 in order to access table *TenKOne* evenly. There is no index on *OneK.tenthous*, but there is a nonunique index on *TenKOne.hundred.*

From Table 4.8 and Figure 4.4, we notice that in this case, the join query is constantly worse than the equivalent TypeN query. In the worst case, the join query can be 16% slower than the TypeN query. The query planner indicates that both queries use *nested-iteration* method except that for query 3.3, the result from the inner block has to be sorted before executing the nested loop. Usually a nested-iteration method is more efficient when the table in the inner loop fits into the main memory cache. But for both query 3.3 and query 3.4, the larger table *TenKOne* is in the inner loop since this table has an index to be used. This shows us that the Oracle optimizer can make use of index information. But actually, putting the indexed table in the inner loop in a nested-iteration method is not always beneficial if this indexed table is large and the table in the outer loop can fit into the main memory cache as is the case here. After sorting the return tuples from the inner block in query 3.3, the sorting result is at most 100 integers, so most of the main memory cache can be occupied by the inner loop table *TenKOne*. But in join processing, *OneK*'s taking up additional space may cause extra disk I/Os for accessing *TenKOne*. However, this performance data may be due to the implementation of our particular Oracle optimizer.

Query	*Outer*	*Inner*	*Number of Return Tuples*						
	Table	*Table*	*100*	*1000*	*2000*	*4000*	*6000*	*8000*	*10000*
TypeN	Index		1.32	7.86	15.12	29.70	44.29	58.37	73.10
Join	Index		1.39	8.10	16.57	34.41	50.87	67.35	84.42

Table 4.8: Timing Data(Sec.) for TypeN/Join Queries (3.3/3.4) with a Table Fit into Main Memory Cache

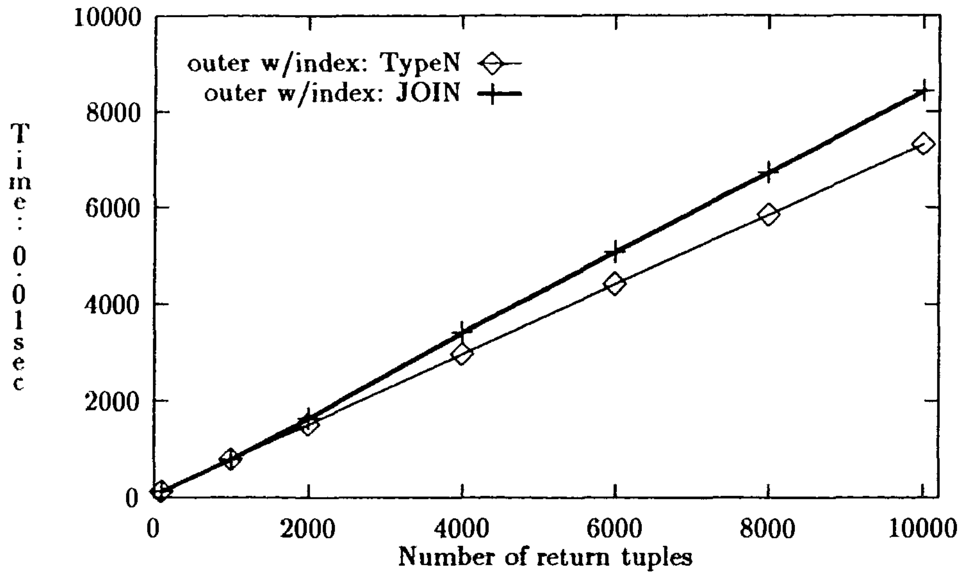

Figure 4.4: TypeN/Join Queries (3.3/3.4) with a Table Fit into Main Memory Cache

Thus, in the case where none of the tables can fit into cache, a TypeN query should be transformed into the equivalent join query in order to obtain better performance.

4.3.4 TypeJ Queries

A TypeJ query can also be transformed into a logically equivalent join query. The queries used for the test are

```
4.1:

        SELECT  even100
        FROM    TenKOne
        WHERE   value1 < value3 AND
                value1 IN
            (   SELECT  value2
                FROM    TenKTwo
                WHERE   value2 < value3 AND
                        TenKTwo.two <= TenKOne.odd100 )
4.2:

        SELECT  TenKOne.even100
        FROM    TenKTwo, TenKOne
        WHERE   TenKOne.value1 < value3            AND
                TenKTwo.value2 < value3            AND
                TenKTwo.value2 = TenKOne.value1    AND
                TenKTwo.two <= TenKOne.odd100.
```

In this test, *value3* is actually the number of return tuples of each query since the join condition *TenKOne.two* <= *TenKTwo.odd100* is always true and *value1, value2* are the keys for the corresponding tables.

For the TypeJ query, if there are indexes *tenk1unik, tenk2unik* available for the evaluation on both the outer block and the inner block respectively, the access plan on our system is

```
FILTER
    TABLE ACCESS BY ROWID TenKOne
      INDEX RANGE SCAN tenk1unik
    TABLE ACCESS BY ROWID TenKTwo
      INDEX UNIQUE SCAN tenk2unik
```

The access plan for the logically equivalent join query is

```
NESTED LOOPS
    TABLE ACCESS BY ROWID TenKOne
      INDEX RANGE SCAN tenk1unik
    TABLE ACCESS BY ROWID TenKTwo
      INDEX UNIQUE SCAN tenk2unik
```

If no index is available, a *full-table-scan* will be used. The *FILTER method* is another name for the nested-iteration method for nested queries. For each tuple in the outer table which satisfies the selection condition, the entire table in the inner block is searched. If the nested predicate is true, the corresponding values from the tuples of the outer table are output. Thus, the performance of TypeJ query with both indexes

Query	*Outer Table*	*Inner Table*	*Number of Return Tuples* 100	1000	2000	4000	6000	8000	10000
TypeJ		Index	8.76	23.98	38.63	68.27	97.84	130.76	162.71
Join			8.81	23.96	37.99	67.46	98.68	129.62	164.35
TypeJ	Index	Index	0.94	11.70	23.67	47.61	71.11	95.11	118.40
Join			0.93	11.53	23.52	47.31	70.68	93.90	117.90
TypeJ	Index		220.00	2254.44	–	–	–	–	–
Join			5.34	14.43	23.74	41.94	59.92	80.13	100.10
TypeJ			235.88	2313.49	–	–	–	–	–
Join			10.42	17.39	25.33	48.27	70.44	83.62	101.20

Table 4.9: Timing Data(Sec.) for TypeJ/Join Queries (4.1/4.2)

should be the same as the corresponding Join query if the same index information is available. We also chose four cases to test the effect of an index on the query: inner and outer tables with and without index used by the optimizer.

From Table 4.9, we can see that if there is an index which would be used by the optimizer in the inner query block, no matter whether or not the outer block has an index, the difference between those corresponding queries is fairly small. This is due to the fact that in both cases, the optimizer selects the same access plan.

If there is no index in the inner block, a huge performance difference is detected between logically equivalent queries. If there is no index in either inner or outer blocks, the TypeJ query needs 235.88 seconds when *value3 = 100* , but the equivalent join query is 23 times faster (requiring only 10.42 seconds). This difference is because the nested-iteration method must be used for a TypeJ query, and both large tables must be full-table-scanned. However, a join query is processed with the merge-sort method which is much more efficient. If only the outer block has an (unclustered, unique) index, we found that the TypeJ query requires 220 seconds when *value3 = 100*, but the join query is 41 times faster, needing only 5.34 seconds (see Table 4.9). This is due to the fact that TypeJ query execution always follows the SQL syntax, putting the inner table at the inner loop. If there is no index available for the inner table, this is very inefficient. But the join query can put the indexed table at the inner loop and obtain a performance benefit, especially when a small part of that indexed table is accessed. When a large part of the indexed table is accessed, the TypeJ query would require more time. Even though accessing index involves an overhead, the join query is still expected to perform better than the TypeJ query.

Generally speaking, if there is no index on the inner table which can be used by the optimizer, the join query is much more efficient than the TypeJ query. Only in this case will the transformation from TypeJ to Join achieve a big benefit. If there is an index of the inner block table available for the evaluation, there is no point in transforming a TypeJ query since both forms use the same algorithm on our Oracle system.

4.3.5 TypeJA Queries

The queries used for testing are

5.1:

```
        SELECT  even100
        FROM    OneK
        WHERE   value1 < value3  AND
                value1 <=
            (   SELECT  MAX( value2 )
                FROM    TenKTwo
                WHERE   TenKTwo.value2 < value3  AND
                        TenKTwo.two <= OneK.odd100 ),
5.2:
        SELECT  OneK.even100
        FROM    OneK, TMP2
        WHERE   OneK.value1 <= TMP2.max  AND
                OneK.odd100 = TMP2.odd100,


        Where:  TMP1( odd100 ) = (
                    SELECT  DISTINCT  odd100
                    FROM    OneK
                    WHERE   value1 < value3 )
and
                TMP2( odd100, max ) = (
                    SELECT  TMP1.odd100, MAX( TenKTwo.value2 )
                    FROM    TMP1, TenKTwo
                    WHERE   TenKTwo.value2 < value3  AND
                            TenKTwo.two <= TMP1.odd100
                    GROUP BY  TMP1.odd100 ).
```

The join condition *TenKTwo.two* <= *OneK.odd100* is always true, and *value1* and *value2* are key attributes for the corresponding tables and the values of them are consecutive integers from 0 on up. Therefore, the value of *value3* is the actual number of return tuples from each query.

If there are indexes in both the inner and the outer blocks which can be used by the optimizer, the access plan of query 5.1 is

```
FILTER
    TABLE ACCESS BY ROWID OneK
      INDEX RANGE SCAN onekunik
    SORT GROUP BY
      TABLE ACCESS BY ROWID TenKTwo
        INDEX RANGE SCAN tenk2unik.
```

For each tuple in table *OneK* found by *INDEX RANGE SCAN* which satisfies that selection condition, the entire table *TenKTwo* is searched by *INDEX RANGE SCAN* for those tuples which satisfy all of the selection conditions and the join conditions. The maximum of *value2* value in those tuples is found by *SORT GROUP BY.* If the *value1* value of the tuple in *OneK* is not greater than the maximum *value2* value, then the value of *even100* of that tuple is output. In the situation where no index exists, the corresponding table will be *FULL TABLE SCAN*ned instead of *INDEX RANGE SCAN*ned.

The transformation algorithm creates some intermediate tables. But generally speaking, the intermediate tables are fairly small in size because the attributes in the intermediate tables are the attributes refered to in the predicates in the WHERE

Query	*Outer Table*	*Inner Table*	*Number of Return Tuples* 100	200	300	400	600	800	1000
TypeJA		Index	10.13	40.72	182.41	302.57	–	–	–
Join			9.42	16.58	38.20	48.95	72.80	95.61	117.43
TypeJA			492.36	948.33	1433.82	–	–	–	–
Join			11.75	15.18	19.30	22.31	28.76	35.35	41.86
TypeJA		Index	9.29	34.45	164.87	296.70	–	–	–
Join	Index		12.23	23.91	49.15	65.35	96.84	128.34	157.98
TypeJA			484.76	923.52	1378.47	–	–	–	–
Join			14.16	22.21	29.91	37.93	53.41	68.29	83.42

Table 4.10: Timing Data(Sec.) for TypeJA/Join Queries (5.1/5.2)

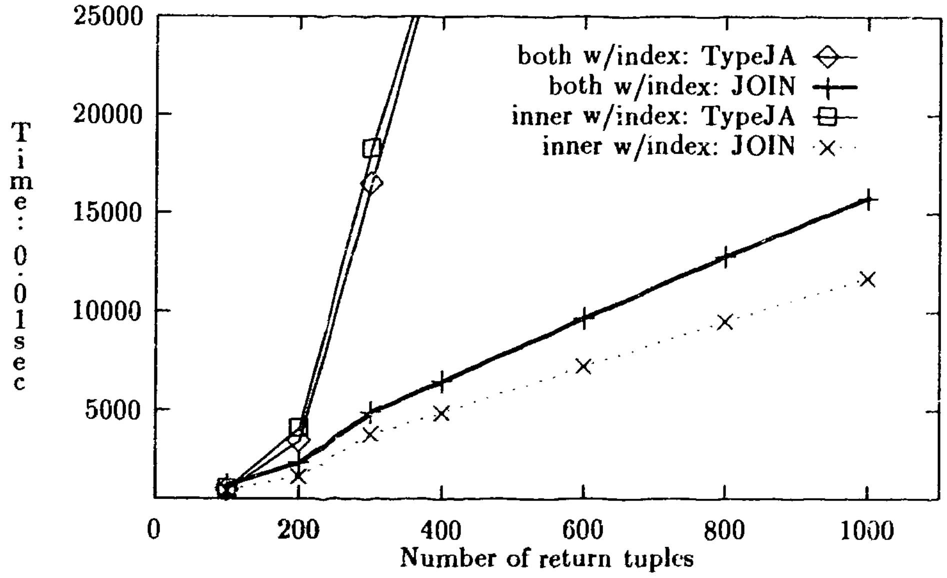

Figure 4.5: TypeJA/Join Queries (5.1/5.2) with Different Index Information

clause. Those attributes are usually quite few in number. Thus, the intermediate tables can usually fit in the main memory cache and the join should be very efficient. Of course, the creation of the intermediate tables takes some time as well. We created two temporary tables for query 5.2.

From Table 4.10 and Figure 4.5, we observe that the join query is much more efficient than the TypeJA query in all the index cases. Thus, a TypeJA query should always be transformed into its logically equivalent join form according to these results.

4.3.6 EXISTS Extension

The query block after EXISTS may or may not have a join predicate which connects the two query blocks and EXISTS may be expressed by an aggregate function *COUNT*. This makes the equivalent queries have either TypeA nesting or TypeJA nesting. Thus, there are four testing queries.

6.1:

```
        SELECT  even100
        FROM    TenKOne
        WHERE   EXISTS
            (   SELECT  value2
                FROM    TenKTwo
                WHERE   value2 < value3 );
```

6.2:

```
        SELECT   even100
        FROM     TenKOne
        WHERE    0 <
            (    SELECT   COUNT( value2 )
                 FROM     TenKTwo
                 WHERE    value2 < value3 );
```

6.3:

```
        SELECT   even100
        FROM     TenKOne
        WHERE    EXISTS
            (    SELECT   value2
                 FROM     TenKTwo
                 WHERE    TenKTwo.value2 < value3   AND
                          TenKTwo.two <= TenKOne.odd100 );
```

6.4:

```
        SELECT   TenKOne.even100
        FROM     TMP1, TenKOne
        WHERE    TMP1.count > 0    AND
                 TenKOne.odd100 = TMP1.odd100;


        Where:
            TMP2( odd100 ) =
                 SELECT   DISTINCT   odd100
                 FROM     TenKOne;
```

```
TMP3( two, value2 ) =
    SELECT  two, value2
    FROM    TenKTwo
    WHERE   value2 < value3;
TMP1( odd100, count ) =
    SELECT  TMP2.odd100, COUNT( TMP3.value2 )
    FROM    TMP2, TMP3
    WHERE   TMP2.odd100 >= TMP3.two (+)
    GROUP BY  TMP2.odd100.
```

Query 6.1 is logically equivalent to query 6.2 of TypeA, and query 6.3 is equivalent to query 6.4 of join after the transformation from TypeJA by three temporary tables. For all of those four queries, the number of return tuples are either zero (if *value3* is zero) or 10,000 (if *value3* is greater than zero). According to the transformation algorithm, an outer join is used for the query 6.4 when the temporary table *TMP1* is created. For all of the four queries, we tested two cases for the inner query block with and without an index.

From Table 4.11, we can see that in all of the cases, there is a big performance increase between *value3 = 0* and *value3 = 1*. This is due to the fact that when *value3 = 0*, no tuple returns at all, but when *value3 = 1*, the entire table is returned. This big difference is due to the *FETCH* operation in Pro*C[7]. But once the entire table is

[7]FETCH is a Pro*C operation which gets the rows one by one from the answer set of a query which returns multiple rows.

		Number of Return Tuples from Inner Block				
Query	*Index*	*0*	*1*	*100*	*1000*	*10000*
EXISTS (6.1)	Yes	0.60	68.74	68.42	68.72	69.18
TypeA (6.2)	Yes	0.60	69.23	68.87	68.81	69.56
EXISTS (6.1)	No	4.70	70.98	69.73	68.80	68.63
TypeA (6.2)	No	4.58	73.54	74.15	73.77	74.83
EXISTS (6.3)	Yes	16.27	83.80	84.26	83.83	83.72
Join (6.4)	Yes	10.51	92.14	92.81	119.85	382.97
EXISTS (6.3)	No	46870.84	18677.62	-	-	-
Join (6.4)	No	15.37	95.25	98.55	127.71	361.31

Table 4.11: Timing Data(Sec.) for EXISTS and Related Queries

returned, the timing data is consistent for all the cases.

If there is an index in the inner table available for the evaluation, the two queries have almost the same performance data. In this case, the evaluation of the inner query block is completed simply by scanning the index data instead of accessing the table data through the index, and the whole index data can fit into the cache. Thus, the performance difference is fairly small. While the EXISTS query just scans the index until the first qualified data is found, the TypeA query has to go through the entire index in order to *count*. So, we observe that the TypeA query is slightly slower than the EXISTS query.

But if no index is available, both queries must access the table data. Thus, we see a performance difference between the two queries. While the EXISTS query is looking for the first qualified tuple in the table, the TypeA query has to count the entire table. Thus, the TypeA query is even slower.

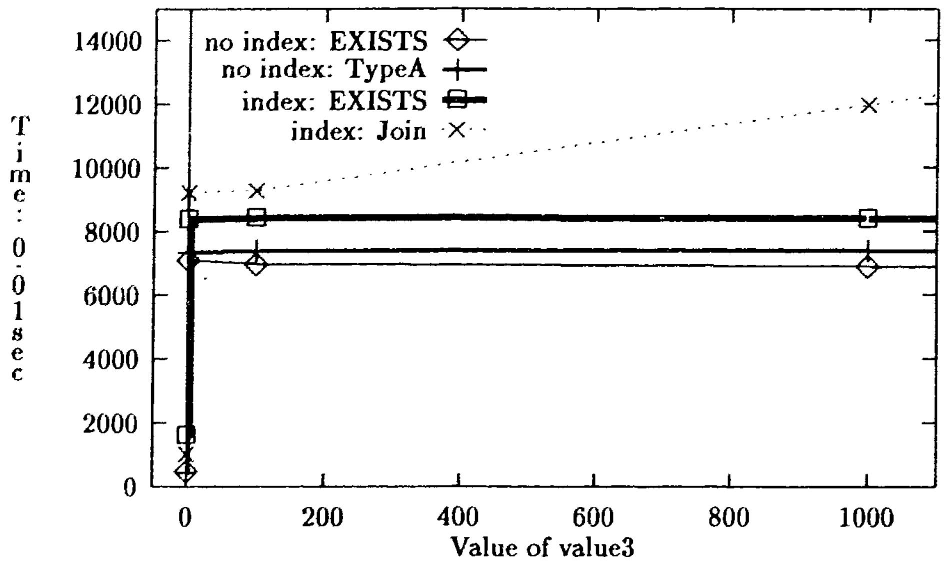

Figure 4.6: EXISTS Queries (6.1 and 6.3) with Different Index Information

For query 6.3, if there is an index in the inner table, the access plan would be

```
FILTER
    TABLE ACCESS FULL TenKOne
    TABLE ACCESS BY ROWID TenKTwo
      INDEX RANGE SCAN tenk2unik.
```

If no index is available, *TenKTwo* is full table scanned instead of using the index. An EXISTS query still tries to find out the first tuple, thus the timing data is the same if there is at least one tuple to make the EXISTS condition true. But the timing data is increased with the number of tuples involved in the join query. If there is an index available, the join query would be slower than the EXISTS query if the EXISTS condition is true. If there is no index at all, the join query is more efficient since, in this case, the EXISTS query has to pass through the very inefficient nested looping. This is the only case that an EXISTS query should be transformed into a join query by introducing three temporary tables. Notice that if there is no index available, the EXISTS query, which returns nothing, takes a longer time than that which returns the entire table of 10,000 tuples. This is because the query must scan the entire table in order to return nothing at all instead just to find out the first tuple which makes that EXISTS condition true to return the entire table.

4.3.7 ANY/ALL Extensions

By consulting the query planner, we know that for the same kind of ANY and ALL queries, the query planner chooses the same access plan which means that if the only difference between two SQL queries is the difference between ANY and ALL at the same place in the queries, then the access plan is very similar. For example, query

```
SELECT  even100
FROM    OneK
WHERE   value1 >= value3  AND
        value1 > ALL
     (  SELECT  value2
        FROM    TenKTwo
        WHERE   value2 < value3 )
```

has a similar access plan as query

```
        SELECT  even100
        FROM    OneK
        WHERE   value1 >= value3  AND
                value1 > ANY
            (   SELECT  value2
                FROM    TenKTwo
                WHERE   value2 < value3 ).
```

Thus, for the test, we only choose the ANY extension.

Depending upon whether or not there is a join predicate in the inner query block, an ANY query may be transformed into a TypeA or TypeJA query. Therefore, the test queries are

```
7.1:
        SELECT  even100
        FROM    TenKOne
        WHERE   value1 < value3  AND
                value1 <= ANY
            (   SELECT  value2
                FROM    TenKTwo
                WHERE   value2 < value3 );
7.2:
        SELECT  even100
        FROM    TenKOne
        WHERE   value1 < value3  AND
                value1 <=
            (   SELECT  MAX( value2 )
                FROM    TenKTwo
                WHERE   value2 < value3 );
```

```
7.3:
        SELECT  even100
        FROM    OneK
        WHERE   value1 < value3  AND
                value1 <= ANY
            (   SELECT  value2
                FROM    TenKTwo
                WHERE   TenKTwo.value2 < value3  AND
                        TenKTwo.two <= OneK.odd100 );
```

Among them, query 7.1 and 7.2, query 7.3 and 5.1 are logically equivalent. The value of *value3* is the number of return tuples from the queries. We tried all of the four cases for index information: inner and outer query block with and without an index.

The optimizer chooses the same access plan for query 7.1 and query 7.2 except for query 7.2, *TenKTwo* has to be sorted before the filter algorithm. If both the inner block and the outer block has an index, the access plan for query 7.1 is

Query	*Outer Table*	*Inner Table*	*Number of Return Tuples* 100	1000	2000	4000	6000	8000	10000
<=ANY			15.66	58.38	50.78	66.86	84.44	119.77	109.13
TypeA			10.56	17.11	23.91	37.34	51.13	64.58	78.46
<=ANY	Index		10.37	56.63	48.84	73.29	93.81	127.74	121.34
TypeA			5.75	13.28	22.41	38.88	55.52	72.61	89.74
<=ANY		Index	5.30	12.50	20.23	36.25	50.84	66.42	82.41
TypeA			5.25	11.59	18.46	31.87	45.25	58.72	72.15
<=ANY	Index	Index	0.93	9.59	19.27	38.12	57.62	76.37	95.79
TypeA			0.81	8.73	17.44	34.66	51.91	69.41	86.69

Table 4.12: Timing Data(Sec.) for <=ANY/TypeA Queries (7.1/7.2)

```
FILTER
    TABLE ACCESS BY ROWID TenKOne
      INDEX RANGE SCAN tenk1unik
    INDEX RANGE SCAN tenk2unik.
```

The access plan for query 7.2 is

```
FILTER
    TABLE ACCESS BY ROWID TenKOne
      INDEX RANGE SCAN tenk1unik
    SORT GROUP BY
      INDEX RANGE SCAN tenk2unik.
```

From Table 4.12, we notice that in all of the four index cases, the TypeA query is always more efficient than the corresponding ANY query. Even though the result from the inner query block of query 7.2 implies the need to sort, this sorting may not

Query	*Outer Table*	*Inner Table*	*Number of Return Tuples* 100	200	300	400	600	800	1000
<=ANY	Index	Index	0.96	2.10	3.29	4.32	6.47	8.53	10.61
Join			12.23	23.91	49.15	65.35	96.84	128.34	157.98
<=ANY		Index	1.41	2.37	3.42	4.39	6.25	8.13	10.19
Join			9.42	16.58	38.20	48.95	72.80	95.61	117.43
<=ANY	Index		10.94	19.12	25.63	15.70	49.85	32.75	53.17
Join			14.16	22.21	29.91	37.93	53.41	68.29	83.42
<=ANY			11.26	19.69	26.54	16.19	50.12	32.95	53.49
Join			11.75	15.18	19.30	22.31	28.76	35.35	41.86

Table 4.13: Timing Data(Sec.) for <=ANY/Join Queries (7.3/5.1)

be required since the index is sorted already. Since there is only one value returned from the inner block due to the aggregate function, the selection for the outer block is very straight forward. But because in query 7.1, the result from the inner block is multi-valued in most of cases, finding the first one which satisfies the <=ANY condition may require searching the entire index. Thus, the selection for the outer block has to be based on a set instead of one value. This takes longer time. Thus, in this case, a <=ANY query should be transformed into the TypeA query.

If there is an index available from the inner table, the <=ANY query is always more efficient than the join query, often dramatically so (see Table 4.13). But if there is no index for the inner table, or the index attributes are different from the attribute used for the *SELECT* set, the <=ANY performance will depend on the tuple order of the inner table. In this case, the timing data is not regular. If the outer table has an index, but the inner table does not, the join query is constantly slower than the <=ANY query. Generally speaking, for a TypeJA ANY query without an index from

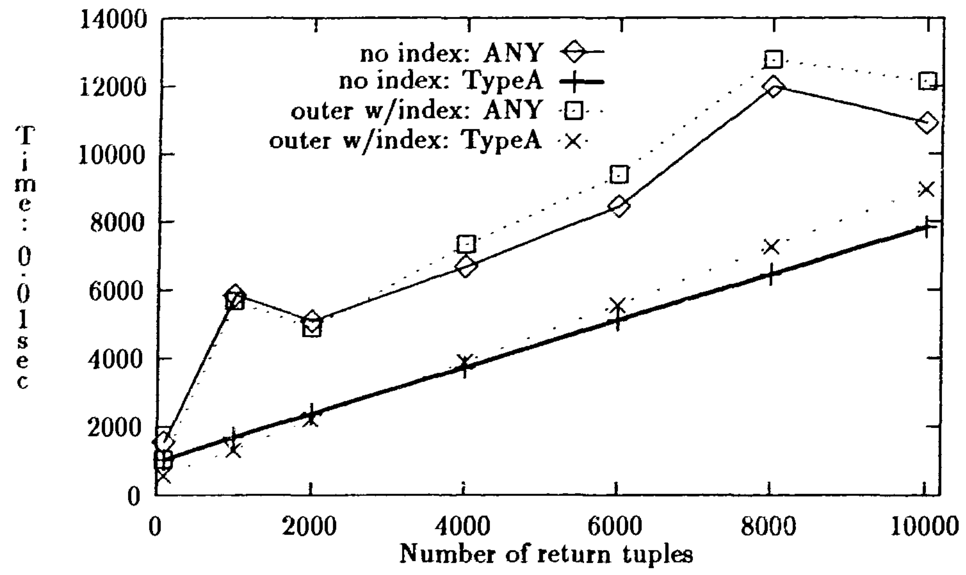

Figure 4.7: <=ANY/TypeA Queries (7.1/7.2) with different index information

the inner table, the transformation can bring minimal efficiency.

It is interesting that the timing data for the <=ANY query does not display the usual linear distribution when there is no index available from the inner table or the index attributes are different from the attribute used to form the *SELECT* set (see Figure 4.7). This is because the member of the list comprising the returned values from the inner block has the same order as the order of the tuple in the inner table. Thus, as more numbers are put into that list, the largest number may appear at the very beginning of that list. Since the ANY algorithm finds the first one, if the number at the beginning is large enough, the evaluation does not need to proceed through the values after that number. This makes the comparison with the list member much quicker. The same thing happened in the <=ANY/Join case (see Figure 4.8). From

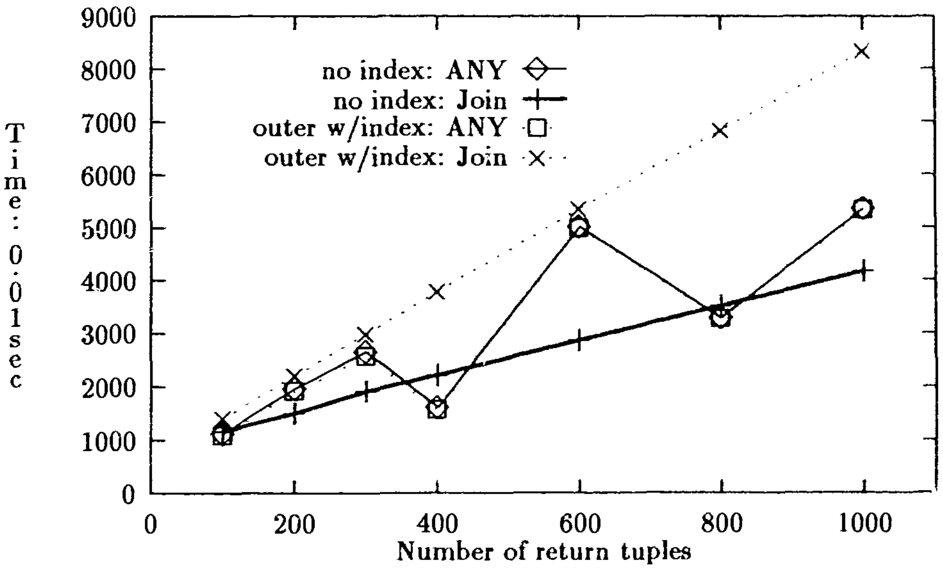

Figure 4.8: <=ANY/Join Queries (7.3/5.1) with different index information

this fact, we can see that the performance of ANY query is related to the tuple order in the table. The ANY query is very similar to the EXISTS query since both of them try to find the first value which satisfies the selection condition. After finding the first one, the query processing should stop in order to get better time efficiency.

We would not expect the ALL query also to find the first tuple in a table. Thus we anticipate that an ALL query should take longer time than the ANY query achieved by replacing the *ALL* in the ALL query by *ANY*. From our above results, we know that TypeA queries are more efficient than ANY queries, and thus a TypeA ALL query should be less efficient than the TypeA query. If an ALL query may be transformed into a TypeJA query, then they have exactly the same access plan according to the query planner. Thus, a TypeJA ALL query should be transformed into the equivalent join query by algorithm TypeJA.

The =ANY operator is logically equivalent to the IN operator, theoretically. Thus, any =ANY operator can be translated to IN no matter what kind of query it is, TypeN or TypeJ. The query planner indicates that for these two kinds of queries, the two operators yield exactly the same access plan. Therefore, it is not necessary to make any transformation between them.

4.4 Summary

The conclusions we make from the above tests are experimentally verified based on our Oracle system and the Wisconsin benchmark database. Some of them, such as the first three conclusions, should be suitable to most of the RDBMSs. We summarize the following:

1. An index does not always guarantee more efficiency. At times, an index can degrade the performance due to extra accesses to index blocks. For Oracle, if a table can not fit into the main memory cache, and more than 25% of the tuples are retrieved from it, an entire table scan is more efficient than accessing the table through indexes.

2. From a performance point-of-view, for the same index situation, the predicate order in single-block queries, e.g., single-table or join queries, does not influence the query performance because the same access plan is used. Since most of the current query optimizers can optimize this kind of queries very well with available index information, no matter what order the predicates are in, an

automatic SQL query generation system can disregard predicate order.

3. There are two access plans for a join query: *nested-loop method* and *merge-sort method*. Which one is used for a query depends on the indexes. The use of an index can make join queries more efficient if the join tables can not fit into the main memory cache and a small proportion of the tables is accessed. But it is not the case that the more the indexes, the more efficient the queries. If some condition testing can be done by only accessing the indexes instead of accessing the entire table through the index, the query can be much more efficient, especially when a large proportion of the indexed attributes needs to be accessed. This is due to the fact that normally an index is small enough to fit into the main memory cache. But accessing a large proportion of a table through an index will degrade the join performance.

4. On our Oracle system, a TypeN nesting can be processed by the merge-sort method or the nested-iteration method. Comparing a TypeN query with its logically equivalent join form, we found from our experiments that a TypeN query should be transformed into the join query in order to obtain better performance if none of the tables fit into the main memory cache. If some tables do fit into the cache, the TypeN query could be more efficient.

5. A TypeJ query is always evaluated by the nested-iteration method. It is always worthwhile to transform a TypeJ nested query into its logically equivalent join form. From our tests, we can see that there is a huge performance difference between these two kinds of queries and the join query is always better, especially when the inner table of the TypeJ query does not have any index. For a TypeJ query, the SQL syntax decides the evaluation plan, but for a join query,

the optimizer can choose the inner loop table for the nested-iteration method according to available indexes.

6. A TypeJA nesting can only be evaluated by the nested-iteration method on our Oracle system. A TypeJA query could be much less efficient than the equivalent join query, even though some temporary tables have to be established in order to finish the join query. The temporary tables are usually very small and can fit into the cache, thus making the final join operation very efficient. Therefore, a TypeJA query should be transformed into its join query according to the algorithm provided.

7. Our tests demonstrate that if an EXISTS can be transformed into a TypeA query, that EXISTS query can never be less efficient than the TypeA query. If an EXISTS query can be transformed into a join query, then that join query will be much more efficient if there is no index in the inner table of the EXISTS query. The processing of EXISTS queries uses a *find-first* algorithm, thus the query has an almost constant performance, but the performance of the equivalent join query would increase linearly with the number of return tuples. If the *find-first* algorithm is used, the query processing should stop after finding the first one in order to get better performance. If there is no index available for the inner table, then this is the only case in which an EXISTS query should be transformed into a join query, if possible.

8. From a performance point-of-view, the set inclusion operator IN always has the same efficiency as the =ANY operator. This is due to the fact that the query planner of our system uses the same access plan for both of them. For other kinds of ANY queries, if an ANY query can be transformed into an equivalent

TypeA query, then the TypeA query can always be more efficient than the ANY query in our testing. But if an ANY query can be transformed into a join query with *Algorithm TypeJA*, in most cases, we cannot derive a performance benefit at all. This is because the evaluation of the ANY queries uses a *find-first* algorithm; and it is faster in our tests. Because of the *find-first* algorithm, the performance of the ANY query is sometimes related to the order of tuples concerned. The ALL query should always be transformed into its equivalent form. And if the *find-first* algorithm is used, the query processing should stop after finding the first one in order to get better performance.

CHAPTER 5

A Transformation for General Nested Queries

A generally more effective strategy for evaluating a nested query of arbitrary depth and complexity is to transform it to its join form and have the optimizer determine an optimal set of algorithms and access paths for evaluating it. Our experiment indicates to us that the join query after transformation is more efficient in most of the cases we considered. It appears that the equivalence-transformation approach developed in the previous chapters may be adopted as the foundation for an optimizer of SQL-like queries and the nested-iteration method may then be used to augment the performance of the optimizer for the rather special situations for which the nested-iteration method is more efficient.

For a more general nested query, all different kinds of nesting can be mixed up, and for a general TypeJA nesting, the aggregate function and the join predicate may appear at any level of nesting, and not necessarily at the same level. While the algorithm for a TypeN/J nesting in Figure 3.1 may work as well if the depth of nesting is greater than one, the algorithm developed in Figure 3.2 can only work with the simplest TypeJA nested queries. Thus, the algorithm in Figure 3.2 needs to be extended. A direct postorder recursive algorithm which works for general TypeA, TypeN, TypeJ, TypeJA nested queries was developed in [12]. This algorithm is presented here.

Figure 5.1 illustrates the pseudocode for this algorithm. The parameter *QueryBlk* is a pointer to an SQL query block, possibly with descendant inner query blocks nested within it. Initially, *QueryBlk* points to the outermost query block of a query. After calling this procedure, the original query should be transformed into a single block query, which might be a multiway join query with some new tables created by the transformation algorithms.

Three procedures are called within procedure *nestG()*.

- *nestA(QueryBlk):* evaluate the query block pointed to by *QueryBlk*, replacing it with the resulting constant. This is used for evaluating the inner query block of a TypeA query.
- *nestNJ(QueryBlk):* execute the algorithm in Figure 3.1, which transforms a single-level TypeN or TypeJ nesting pointed to by *QueryBlk* into a join query. This is used for transforming TypeN or TypeJ nested queries.
- *nestJA1(QueryBlk):* execute the first two steps in the algorithm of Figure 3.2,

```
procedure nestG( QueryBlk )
Struct SQLQueryBlk  *QueryBlk;
{
    for ( each nested predicate in the WHERE clause of QueryBlk )
    {
        /* transform the subquery */
        nestG( QueryBlk->InnerBlk );

        /* decide the nesting type for QueryBlk, it's only single-level
         * nesting here
         */
        if ( the SELECT clause of the inner block has an aggregate
             function ) {
            if (inner block has join with table not in its FROM clause){
                /*
                 * TypeJA nesting
                 */
                nestJA1( QueryBlk->InnerBlk );
                nestNJ( QueryBlk );
            }
            else /*
                  * TypeA nesting
                  */
                nestA( QueryBlk->InnerBlk );
        }
        else /*
              * TypeN or TypeJ nesting
              */
            nestNJ( QueryBlk );
    }
    return;
}
```

Figure 5.1: Algorithm for Transformation of General Nested Queries

which creates a temporary table with a GROUP BY operation as specified in that algorithm. This is used for removing the aggregate function in order to transform a TypeJA query into a TypeJ query. The procedure *nestNJ()* should be called right after this in order to transform the resulted TypeJ nesting into a join form.

Procedure *nestG()* searches down through the nesting levels of a nested query from the outermost query block until it finds the innermost nested query block of each branch. It then examines the inner block to determine the type of nesting present, and transforms the single-level nesting there to join form by calling the appropriate transformation procedures. After this is done for all of the nested predicates in the current block, the recursion then backs up one level and the block is processed in the same way. The procedure keeps doing this backup until the outermost nested block is transformed. Note that procedure *nestJA1()* is similar to the algorithm for the simplest TypeJA queries, but the difference is that the *nestJA1()* is used to transform a TypeJA query into a TypeJ query instead of a join query directly. This is because the transformation to the join query may involve some other tables which appear in the other parts of the initial nested query. After creating the temporary table by *nestJA1()*, the aggregate function is removed by replacing it with a reference to the column in the temporary table which results from the application of the aggregate function. This reduces the TypeJA nesting into TypeJ nesting, and after that, *nestNJ()* is called immediately to finish the job of reducing the TypeJ query to a join form. Any transformation in this algorithm is confined to single-level nesting: the outer block pointed to by *QueryBlk* and the inner block pointed to by *QueryBlk->InnerBlk*.

A nesting can be represented by a tree structure based on the SQL syntax. Each *node* in the tree represents a query block in the initial query, and there is an *edge* between two nodes if the corresponding query blocks of these two nodes are nesting. There are two kinds of edges: *dashed edge* and *solid edge.* A solid edge represents the SQL query syntax. Thus, all the solid edges form a tree according to the syntax. Dashed edges represent join operations between two non-directly-nested query blocks. The outermost query block, e.g., the beginning of the SQL statement, is the *root node* of the tree, and the innermost query blocks are the *leaves.* The label on an edge represents the nesting present between those two blocks connected by the edge.

The following example shows us how to form this query tree and how this recursive approach works. A general nested query is represented in Figure 5.2. The outermost query block is represented by the root node A. Query block B appears in the WHERE clause of A and contains an aggregate function in its SELECT clause; there is a join predicate in block C which involves a table in the FROM clause of block B; block E has two join predicates, one involves a table in block C and another in block A. Since block E is nested in block C by the query syntax, there is a solid edge between E and C, and there is a dashed edge between E and A due to the join between tables in these two blocks. An edge for TypeN nesting is labeled *N* like the edge between B and D. All of the nodes and the solid edges in Figure 5.2 represent the syntactic structure of the query, and thus form into a tree. This example represents a general TypeJA nesting with TypeA, TypeN and TypeJ nesting inside. For this kind of graph representation of nesting, a join dashed edge must span a node containing an aggregate function, like node B in the example, for a TypeJA nesting to be present.

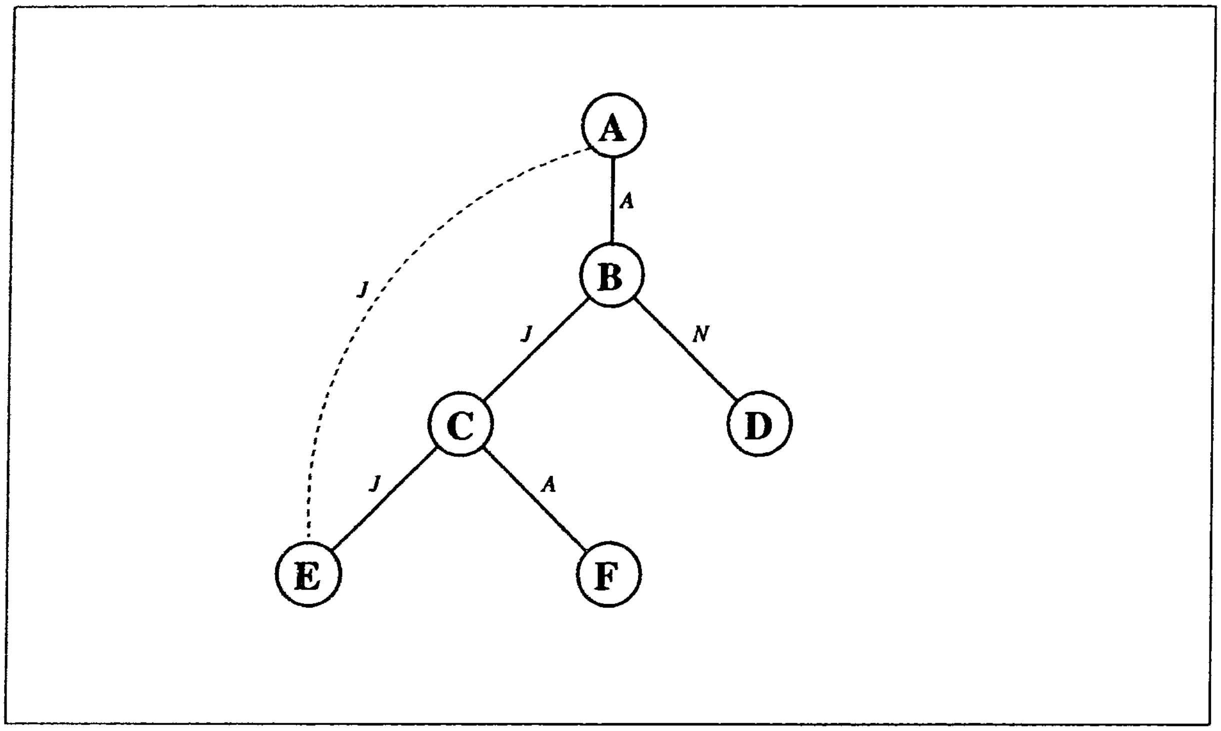

Figure 5.2: An Example of a General Nested Query

Procedure *nestG()* will travel down to E first, backtrack and apply the algorithm to combine C and E into a join block; this moves the join reference to block A in block E initially to block C. Since the nesting between C and F is TypeA, block F is evaluated independently into a constant, thus the nested predicate in C becomes a simple predicate. After this, blocks C and B are combined, and then D and B. Now, the new query block B has an aggregate function in its SELECT clause, and a join predicate which references a table not found in the FROM clause of B, but in block A, thus a TypeJA nesting presents. The TypeJA nesting is first changed into TypeJ nesting by *nestJA1()*, and then to a join query by *nestNJ()*. The execution of *nestNJ()* involves all the tables in nodes B, C, D, E, and a temporary table created by procedure *nestJA1()*.

This algorithm shows a general way to transform a general nested query into its logically equivalent join query. But from here, we observe some drawbacks. The more complicated a query is, the more difficult for people to understand its logical meaning, thus, the less chances for programmers to propose such a complicated query. But the transformation algorithm is still useful if the complicated query is built mechanically by some query generation system such as System X ([19]). Another problem is that the more complex the nesting, the more difficult the evaluation of the performance cost. In addition, a theoretical analysis would be more difficult and more unaccurate because of simplification, and testing would be very time-consuming. Although we have tested the actual performance for some transformations, for more complicated queries, building temporary tables in order to remove nesting may cause extra disk I/Os and extra computation. On the other hand, for complex nested queries, the access plan in contemporary database systems would be heavily dependent on the SQL syntax, and the potential for optimization would be less. Therefore, we still expect that the transformed join query should perform better than the initial nested query. The cost model of this general transformation algorithm still needs to be completed and actual testing needs to be performed.

CHAPTER 6

Conclusions

We observed the performance differences between some logically equivalent SQL queries. Although much work has been done on query optimization, for a particular relational DBMS, the automatic query optimization is less than perfect. A huge performance gap persist between some logically equivalent SQL queries. Some query optimization which is based on some particular transformations and particular index information still needs programmer intervention. Thus, we focused on transforming some SQL queries in order to improve performance.

For a nested SQL query, the nesting structure plays a very important role on choosing the query processing plan. Most of the contemporary query optimizers use the nested-iteration method for the processing of nested queries, even though this is not the only option. For example, the Oracle optimizer may make use of the merge-sort method for some nesting. Furthermore, the nested-iteration method may perform better in particular cases. For join queries on the other hand, the optimizer can choose a processing plan based on its inner-representation form instead of its SQL syntax form, thus taking advantage of index information and other performance-related factors. This is the main reason that most nested queries are less efficient than their logically equivalent join queries. This point further illustrates, as well, that contemporary query optimizers can optimize the join queries better.

In the thesis, we presented some algorithms for the optimization of nested queries by transformation. Since most of the nested queries can be transformed into logically equivalent join queries, more attention should be paid on join query optimization in order to get better time efficiency for most of the SQL queries. Besides index information, some other information, such as the size of main memory cache, should be considered, as well, in join processing.

The performance testing reported herein leads evidence to the fact that the transformation algorithms presented in this thesis are practical and effective. The transformations of TypeJ (if no index in the inner block) and TypeJA queries can obtain much better performance on our Oracle system. How much efficiency may be gained from such transformations will vary on different database systems, depending especially on their query optimizers. Some transformation necessary for an Oracle optimizer may not be necessary for a SYBASE optimizer since they use different methods to process the same query. Thus, in order to make best use of the algorithms in this thesis, the query processing methods used by a particular optimizer need to be considered.

Our testing indicates that index can speed up retrieval sometimes, especially when a small proportion of table is accessed through index or when the retrieval can be accomplished by accessing the index alone. Another role an index plays is that the index information can influence the query optimizer to choose the access plan for a query. This influence can result in a better performance, but at times, some other information, such as the size of main memory cache, should be considered as well.

Theoretical cost models are clear and correct in some sense, but some theoretical cost models for query processing make comparisons based only on some major characteristic, like disk I/Os. In the current systems, disk I/O is only one of the major factors in performance efficiency. It is difficult to obtain an accurate cost model using theoretical analysis since the processing is usually very complicated. Thus, performance testing, especially based on a standard benchmark is appealing. The cost model and actual testing for the transformation algorithm in Chapter 5 are needed. Testing can produce more reliable conclusions for particular databases since a test involves many major factors instead of only one of them. On the other hand, actual testing is usually more time-consuming and less complete.

Database query optimization has been well studied previously. The transformations mentioned in this thesis are based on SQL syntax instead of query semantics. More performance benefit could be obtained if query semantics were considered. For example, TypeJ Query 4.1 produces the same result as the TypeN Query 3.1 because of the attribute value ranges. Judgements about semantic equivalence like above are based on application knowledge such as value ranges and distribution of attributes.

In our testing, transforming a TypeJ query in which only the inner table is indexed into a join cannot obtain performance benefit at all since in this situation, both of them use the same access plan for evaluation. But if the TypeJ query can be transformed into the equivalent TypeN query, and subsequently transformed again into a join query, our testing shows that the query response time may decrease dramatically (in the best case testing, the query response time will decrease from 162.71 seconds to 78.69 seconds, or 51% faster). Thus, query optimization which involves semantic considerations is a potentially rewarding research area.

We found the Wisconsin benchmark database inadequate for our testing purposes in that it did not allow for systematic scaling of attribute ranges and values. For example, it should be possible to model the same fixed selectivities (e.g., 100 tuples) and relative selectivities (e.g., one percent relation cardinality) for different database sizes. Sometimes, it is very difficult to do so, especially when other factors, like query selectivity and index information, need to be considered together. Therefore the Wisconsin benchmark database needs to be improved.

Appendix A

List of SQL Queries for Performance Testing

All of the SQL queries used in our testing are listed here. The testing was performed on Oracle RDBMS Version 6.1. Queries were executed from a Pro*C program. The *TABLE, value, value1, value2* and *value3* in this list are variables. In our testing, they were replaced by the corresponding table names, integer values or attribute names. See Chapter 4 for explaination of each query.

1. Single-Table Queries

```
1.1:    SELECT  even100
        FROM    TABLE
        WHERE   odd100 < 100    AND
                two < 2         AND
                unique1 < value
```

```
1.2:

        SELECT  even100
        FROM    TABLE
        WHERE   unique1 < value AND
                odd100 < 100    AND
                two < 2
```

2. Join Queries

```
2.1(Two-way-join):

        SELECT  TenKOne.even100
        FROM    TenKTwo, TenKOne
        WHERE   TenKTwo.value2 < value3          AND
                TenKTwo.value2 = TenKOne.value1

2.2(Three-way-join):

        SELECT  TenKOne.even100
        FROM    OneK, TenKOne, TenKTwo
        WHERE   OneK.hundred < value             AND
                TenKOne.unique2 < value          AND
                TenKTwo.hundred < value          AND
                OneK.hundred = TenKOne.unique1   AND
                OneK.hundred = TenKTwo.hundred
```

3. TypeN Queries

3.1:

```
SELECT  even100
FROM    TenKOne
WHERE   value1 IN
    (   SELECT  value2
        FROM    TenKTwo
        WHERE   value2 < value3  )
```

3.2:

```
SELECT  TenKOne.even100
FROM    TenKTwo, TenKOne
WHERE   TenKTwo.value2 < value3  AND
        TenKTwo.value2 = TenKOne.value1
```

3.3:

```
SELECT  even100
FROM    TenKOne
WHERE   hundred  IN
    (   SELECT  tenthous
        FROM    OneK
        WHERE   tenthous < value3 )
```

3.4:

```
SELECT  TenKOne.even100
FROM    OneK, TenKOne
WHERE   OneK.tenthous < value3    AND
        OneK.tenthous = TenKOne.hundred
```

4. TypeJ Queries

```
4.1:

        SELECT  even100
        FROM    TenKOne
        WHERE   value1 < value3 AND
                value1 IN
            (   SELECT  value2
                FROM    TenKTwo
                WHERE   value2 < value3 AND
                        TenKTwo.two <= TenKOne.odd100 )
4.2:

        SELECT  TenKOne.even100
        FROM    TenKTwo, TenKOne
        WHERE   TenKOne.value1 < value3             AND
                TenKTwo.value2 < value3             AND
                TenKTwo.value2 = TenKOne.value1     AND
                TenKTwo.two <= TenKOne.odd100
```

5. TypeJA Queries

```
5.1:

        SELECT  even100
        FROM    OneK
        WHERE   value1 < value3  AND
                value1 <=
            (   SELECT  MAX( value2 )
                    FROM    TenKTwo
                    WHERE   TenKTwo.value2 < value3  AND
                            TenKTwo.two <= OneK.odd100 )
```

5.2:

```
SELECT  OneK.even100
FROM    OneK, TMP2
WHERE   OneK.value1 <= TMP2.max  AND
        OneK.odd100 = TMP2.odd100;
Where:  TMP1( odd100 ) = (
            SELECT  DISTINCT  odd100
            FROM    OneK
            WHERE   value1 < value3 )
and
        TMP2( odd100, max ) = (
            SELECT  TMP1.odd100, MAX( TenKTwo.value2 )
            FROM    TMP1, TenKTwo
            WHERE   TenKTwo.value2 < value3  AND
                    TenKTwo.two <= TMP1.odd100
            GROUP BY  TMP1.odd100 )
```

6. EXISTS Extension

6.1:

```
SELECT  even100
FROM    TenKOne
WHERE   EXISTS
```

```
        (   SELECT  value2
            FROM    TenKTwo
            WHERE   value2 < value3 )
```

6.2:

```
    SELECT  even100
    FROM    TenKOne
    WHERE   0 <
        (   SELECT  COUNT( value2 )
            FROM    TenKTwo
            WHERE   value2 < value3 )
```

6.3:

```
    SELECT  even100
    FROM    TenKOne
    WHERE   EXISTS
        (   SELECT  value2
            FROM    TenKTwo
            WHERE   TenKTwo.value2 < value3   AND
                    TenKTwo.two <= TenKOne.odd100 )
```

6.4:

```
    SELECT  TenKOne.even100
    FROM    TMP1, TenKOne
    WHERE   TMP1.count > 0    AND
            TenKOne.odd100 = TMP1.odd100;
    Where:
        TMP2( odd100 ) =
            SELECT  DISTINCT  odd100
            FROM    TenKOne;
        TMP3( two, value2 ) =
            SELECT  two, value2
            FROM    TenKTwo
            WHERE   value2 < value3;
        TMP1( odd100, count ) =
            SELECT  TMP2.odd100, COUNT( TMP3.value2 )
            FROM    TMP2, TMP3
            WHERE   TMP2.odd100 >= TMP3.two (+)
            GROUP BY  TMP2.odd100
```

7. ANY/ALL Extensions

```
7.1:

    SELECT  even100
    FROM    TenKOne
    WHERE   value1 < value3  AND
            value1 <= ANY
        (   SELECT  value2
            FROM    TenKTwo
            WHERE   value2 < value3 )

7.2:

    SELECT  even100
    FROM    TenKOne
    WHERE   value1 < value3  AND
```

```
                value1 <=
            (   SELECT  MAX( value2 )
                FROM    TenKTwo
                WHERE   value2 < value3 )
7.3:
        SELECT  even100
        FROM    OneK
        WHERE   value1 < value3  AND
                value1 <= ANY
            (   SELECT  value2
                FROM    TenKTwo
                WHERE   TenKTwo.value2 < value3  AND
                        TenKTwo.two <= OneK.odd100 )
7.4:
        SELECT  OneK.even100
        FROM    OneK, TMP2
        WHERE   OneK.value1 <= TMP2.max  AND
                OneK.odd100 = TMP2.odd100;
        Where:  TMP1( odd100 ) = (
                    SELECT  DISTINCT  odd100
                    FROM    OneK
                    WHERE   value1 < value3 )
and
                TMP2( odd100, max ) = (
                    SELECT  TMP1.odd100, MAX( TenKTwo.value2 )
FROM     TMP1, TenKTwo
WHERE    TenKTwo.value2 < value3  AND
         TenKTwo.two <= TMP1.odd100
GROUP BY  TMP1.odd100 )
```

Appendix B

A Sample Pro*C Program for Testing

```
/*
 *  This is a Pro*C program used to test SQL TypeN query performance
 *  on the Wisconsin benchmark database.  The Oracle database has to
 *  be connected first, then a SQL query is executed.The UNIX system
 *  call gettimeofday() is used before and after the query execution
 *  in order to obtain the retrieval time. By changing the SQL query,
 *  this program is used in all the tests.
 */

#include    <stdio.h>
#include    <ctype.h>
```

```
#include    <sys/time.h>



/* Definition of host variables. */
EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR    uid[20];         /*  user id for login to ORACLE        */
    VARCHAR    pwd[20];         /*  user passwd for login to ORACLE    */
    int        anything;        /*  used for FETCH operation           */
EXEC SQL END DECLARE SECTION;

EXEC SQL INCLUDE sqlca.h;



main()    {

    /* Define timing variables */
    long                     sec, usec;  /*  second and microsecond  */
    struct    timeval        StartTime;  /*  query start time        */
    struct    timeval        FinishTime; /*  query finish time       */
    struct    timezone       *tzp;       /*  time zone pointer       */


    /* connection to ORACLE */
    strcpy( uid.arr, "qianwu" );
    uid.len = strlen( uid.arr );
```

```
strcpy( pwd.arr, "ericwu" );
pwd.len = strlen( pwd.arr );


EXEC SQL WHENEVER SQLERROR GOTO errexit;
EXEC SQL CONNECT:    uid    IDENTIFIED BY:    pwd;
printf( "\nConnected to Oracle user:%s\n\n", uid.arr );


/* Get the query start time */
tzp = NULL;
if ( gettimeofday( &StartTime, tzp ) == -1 )    {
    printf( "Wrong with gettimeofday().\n" );
    exit( 1 );
}


/* Declare a cursor for a query to return many rows from tables */
EXEC SQL DECLARE cursorpr CURSOR FOR
    SELECT  even100
    FROM    TenKOne
    WHERE   unique1 IN
        (   SELECT  tenthous
            FROM    TenKTwo
            WHERE   tenthous < 2000 );


/* Open the cursor to evaluate the query */
EXEC SQL OPEN    cursorpr;
```

```
    /* Get every row of the query by FETCHING the cursor */
    EXEC SQL WHENEVER NOT FOUND GOTO finish;
    for( ; ; )    {
        EXEC SQL FETCH cursorpr INTO :anything;
    }


errexit: /* SQL error messages */
    printf( "\n%.70s (%d)\n", sqlca.sqlerrm.sqlerrmc, -sqlca.sqlcode );
    exit( 1 );


finish: /* Query execution is successful */

    /* Get the query finish time */
    if ( gettimeofday( &FinishTime, tzp ) == -1 )    {
        printf( "Wrong with gettimeofday().\n" );
        exit( 1 );
    }

    /* Print out the Oracle message */
    printf( "\n%.70s (%d)\n", sqlca.sqlerrm.sqlerrmc, -sqlca.sqlcode );
    /* Close the cursor and commit the transaction */
    EXEC SQL CLOSE   cursorpr;
    EXEC SQL COMMIT WORK RELEASE;

    /* Calculate and print out the timing data */
    sec = FinishTime.tv_sec - StartTime.tv_sec;
    usec = FinishTime.tv_usec - StartTime.tv_usec;
    if ( usec < 0 )    {
        usec += 1000000;
        sec --;
    }
    printf( "\nThe elapsed time in sec.: %ld.%ld \n", sec, usec );

    exit( 0 );
}
```

REFERENCES

[1] Anon, et al.: **A Measure of Transaction Processing Power**, *Datamation*, April 1, 1985, 112-118.

[2] Bitton, D.: **A Retrospective on the Wisconsin Benchmark**, *Readings in Database Systems*, Ed. by M. Stonebraker, *Morgan Kaufmann*, 1988, 280-299.

[3] Bitton, D., DeWitt, D.J., et al: **Benchmarking Database Systems: A Systematic Approach**, *Proc. 9th International Conference on Very Large Data Bases*, Florence, Italy, Nov. 1983, 8-19.

[4] Bitton, D. and Turbyfill, C.: **Design and Analysis of Multi-User Benchmark for Database Systems**, *Technical Report 84-589, Dept. of Computer Science, Cornell University*, Ithaca, New York, Jan. 1984.

[5] Boral, H. and DeWitt, D.J.: **A Methodology for Database System Performance Evaluation**, *Proc. of ACM-SIGMOD Conference on Management of Data*, Boston, 1984, 176-185.

[6] Cardenas, Alfonso F.: **Evaluation and Selection of File Organization – A Model and System**, *Communications of ACM*, Vol. 16, No. 9, Sept. 1983, 540-548.

[7] Chamberlin, D.D., et al: **Support for Repetitive Transaction and Ad Hoc Queries in System R**, *ACM Trans. on Database Systems*, Vol.6, No.1, March 1981, 70-94.

[8] Codd, E.F.: **Extending the database relational model to capture more meaning**, *ACM Trans. on Database System*, Vol.4, No.4, Dec. 1979, 397-434.

[9] Date, C.J.: **A Critique of the SQL Database Language**, *ACM SIGMOD RECORD*, Vol.14, No.3, Nov. 1984, 8-54.

[10] Date, C.J.: **A Guide to the SQL Standard**, *Addison-Wesley*, 1987.

[11] Findelstein, S., Schkolnick, M. and Tiberio, P.: **Physical Database Design for Relational Databases**, *ACM Trans. on Database Systems*, Vol. 13, No. 1, March 1988, 91-128.

[12] Ganski, R.A. and Wong, H.K.T.: **Optimization of Nested SQL Queries Revisited**, *Proceedings of ACM-SIGMOD 1987 International Conference on Management of Data*, San Francisco, May 27-29, 1987, 23-33.

[13] Haberhauer, Franz: **Tutorial: Physical Database Design Aspects of Relational DBMS Implementations**, *Information Systems*, Vol. 15, No. 3, 1990, 375-387.

[14] Jarke, M. and Koch, J.: **Query Optimization in Database Systems**, *ACM Computing Surveys*, Vol.16, No.2, June 1984, 111-152.

[15] Kao, M., Cercone, N. and Luk, W.: **Providing Quality Responses with Natural Language Interface: The Null Value Problem**, *IEEE Trans. on Software Engineering*, Vol. SE-14, No.7, July 1988, 959-984.

[16] Kim, Won: **On Optimizing an SQL-like Nested Query**, *ACM Transactions on Database Systems*, Vol.7, No. 3, Sept. 1982, 443-469.

[17] Korth, H. and Silberschatz, A.: **Database System Concepts**, Second Edition, *McGraw-Hill*, 1991.

[18] Kumar, A. and Stonebraker, M.: **Performance Considerations for an Operating System Transaction Manager**, *IEEE Trans. on Software Engineering*, Vol. SE-15, No. 6, June 1989, 705-714.

[19] McFetridge, P., Hall, G., Cercone, N. and Luk, W.: **Knowledge Acquisition in System X: Natural Language Interface to Relational Databases**, *Proc. of International Computer Science Conference 1988 on Artificial Intelligence: Theory and Applications*, Hong Kong, Dec. 1988, 604-610.

[20] Motzkin, Dalia: **The Design of Optimal Access Paths for Relational Databases**, *Information Systems*, Vol. 12, No. 2, 1987, 203-213.

[21] Oracle Corporation: **SQL*Plus User's Guide**, Version 2.0, 1989.

[22] Oracle Corporation: **ORACLE RDBMS Performance Tuning Guide**, Version 6.0, August 1989, Pages 7-9 to 7-16.

[23] Riet, R., et al: **High-level Programming Features for Improving the Efficiency of a Relational Database System**, *ACM Trans. on Database Systems*, Vol.6, No.3, Sept. 1981, 464-485.

[24] Selinger, P.G., et al: **Access Path Selection in a Relational Database Management System**, *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Boston, May 30-June 1, 1979, 23-34.

[25] Sellis, T.: **Multiple-Query Optimization**, *ACM Trans. on Database Systems*, Vol. 13, No. 1, March 1988, 23-52.

[26] Shneiderman, Ben: **Response Time and Display Rate in Human Performance with Computers**, *ACM Computing Surveys*, Vol.16, No.3, Sept. 1984, 265-285.

[27] Stonebraker, M.: **Readings in Database Systems**, *Morgan Kaufmann*, 1988.

[28] Turbyfill, C.: **Comparative Benchmarking of Relational Database Systems**, *PhD Dissertation*, Technical Report 87-871, Connell University, Sept. 1987.

[29] Welty, C.: **Human Factors Comparison of a Procedural and a Nonprocedural Query Language**, *ACM Trans. on Database Systems*, Vol.6, No.4, Dec. 1981, 626-649.

[30] Yao, S.B., Hevner, A.R. and Young-Myers, H.: **Analysis of Database System Architectures Using Benchmarks**, *IEEE Trans. on Software Engineering*, Vol. SE-13, No.4, June 1987, 709-714.